INTERNATIONAL STANDARD

ISO/IEC 11578

First edition
1996-12-15

Information technology — Open Systems Interconnection — Remote Procedure Call (RPC)

Technologies de l'information — Interconnexion de systèmes ouverts (OSI) — Appel de procédures à distance (RPC)

Reference number
ISO/IEC 11578:1996(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

Section	**1**	**General**	1
	1.1	Scope	2
	1.2	Normative references	3
	1.3	Informative References	4
	1.4	Definitions	5
	1.4.1	Reference model definitions	5
	1.4.2	Application Layer Structure definitions	5
	1.4.3	Service conventions definitions	5
	1.5	Abbreviations	6
	1.6	Portability	8
	1.7	Services and Protocols	9
	1.8	Conformance Requirements	10
Section	**2**	**Introduction to the RPC API**	**13**
	2.1	RPC Programming Model Overview	14
	2.1.1	Client/Server Model	14
	2.1.1.1	Interfaces	14
	2.1.1.2	Remoteness	14
	2.1.1.3	Binding	15
	2.1.1.4	Name Services	15
	2.1.1.5	Resource Models	15
	2.1.1.6	Security Services	16
	2.1.1.7	Server Implementation	16
	2.1.2	Application/Stub/Run-time System	16
	2.1.2.1	RPC Run Time	16
	2.1.2.2	Stubs	16
	2.1.2.3	Application Code	17
	2.2	API Operations	18
	2.2.1	Binding-related Operations	18
	2.2.2	Name Service Operations	18
	2.2.3	Endpoint Operations	18
	2.2.4	Security Operations	19
	2.2.5	Stub Memory Management Operations	19
	2.2.6	Management Operations	19
	2.2.7	UUID Operations	19
	2.3	Binding	20
	2.3.1	Binding Handles	21
	2.3.1.1	Client and Server Binding Handles	21
	2.3.1.2	Obtaining Binding Handles	21
	2.3.2	String Bindings	21
	2.3.3	Binding Steps	22
	2.3.3.1	Server Binding Steps	22
	2.3.3.2	Client Binding Steps	26
	2.3.3.3	Call Routing Algorithms	26
	2.3.4	Binding Methods	29

© ISO/IEC 1996

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

2.4	Name Service Interface	30
2.4.1	Name Service Model	30
2.4.2	Name Syntax Tags	30
2.4.3	Name Service Attributes	31
2.4.3.1	Server Entries	31
2.4.3.2	Group Entries	31
2.4.3.3	Profiles	31
2.4.4	Binding Searches	32
2.4.5	Search Algorithm	33
2.4.6	Name Service Caching	34
2.5	Server Model	35
2.5.1	Server Concurrency and Request Buffering	35
2.5.2	Management Interface	35
2.6	Server Resource Models	36
2.6.1	The Server-Oriented Model	36
2.6.2	The Service-Oriented Model	36
2.6.3	The Object-Oriented Model	36
2.7	Security	37
2.8	Error Handling	37
2.9	Cancel Notification	37
2.10	Stubs	39
2.10.1	IDL to Stub Data Type Mappings	39
2.10.2	Manager EPVs	39
2.10.3	Interface Handles	39
2.10.4	Stub Memory Management	39
2.11	RPC API Routine Taxonomy	40
2.11.1	Binding Operations	40
2.11.2	Interface Operations	40
2.11.3	Protocol Sequence Operations	41
2.11.4	Local Endpoint Operations	41
2.11.5	Object Operations	41
2.11.6	Name Service Interface Operations	42
2.11.6.1	NSI Binding Operations	42
2.11.6.2	NSI Entry Operations	42
2.11.6.3	NSI Group Operations	42
2.11.6.4	NSI Profile Operations	43
2.11.7	Authentication Operations	43
2.11.8	The Server Listen Operation	43
2.11.9	The String Free Operation	43
2.11.10	UUID Operations	43
2.11.11	Stub Memory Management	44
2.11.12	Endpoint Management Operations	44
2.11.13	Name Service Management Operations	44
2.11.14	Local Management Services	45
2.11.15	Local/Remote Management Services	45
2.11.16	Error Messages	45
Section 3	**RPC API Manual Pages**	**47**
3.1	RPC Data Types	47
3.1.1	Unsigned Integer Types	47
3.1.2	Signed Integer Type	47
3.1.3	Unsigned Character String	47
3.1.4	Binding Handle	48
3.1.5	Binding Vector	49
3.1.6	Boolean Type	49
3.1.7	Endpoint Map Inquiry Handle	49
3.1.8	Interface Handle	50
3.1.9	Interface Identifier	50
3.1.10	Interface Identifier Vector	51
3.1.11	Manager Entry Point Vector	51

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.12 Name Service Handle 51
3.1.13 Protocol Sequence String 52
3.1.14 Protocol Sequence Vector 52
3.1.15 Statistics Vector 53
3.1.16 String Binding 53
3.1.17 String UUID 54
3.1.18 UUIDs 54
3.1.19 UUID Vector 54
rpc_binding_copy() 56
rpc_binding_free() 57
rpc_binding_from_string_binding() 58
rpc_binding_inq_auth_client() 59
rpc_binding_inq_auth_info() 61
rpc_binding_inq_object() 63
rpc_binding_reset() 64
rpc_binding_server_from_client() 65
rpc_binding_set_auth_info() 66
rpc_binding_set_object() 68
rpc_binding_to_string_binding() 69
rpc_binding_vector_free() 70
rpc_ep_register() 71
rpc_ep_register_no_replace() 74
rpc_ep_resolve_binding() 76
rpc_ep_unregister() 78
rpc_if_id_vector_free() 79
rpc_if_inq_id() 80
rpc_mgmt_ep_elt_inq_begin() 81
rpc_mgmt_ep_elt_inq_done() 84
rpc_mgmt_ep_elt_inq_next() 85
rpc_mgmt_ep_unregister() 87
rpc_mgmt_inq_com_timeout() 89
rpc_mgmt_inq_dflt_protect_level() 90
rpc_mgmt_inq_if_ids() 91
rpc_mgmt_inq_server_princ_name() 93
rpc_mgmt_inq_stats() 95
rpc_mgmt_is_server_listening() 97
rpc_mgmt_set_authorization_fn() 99
rpc_mgmt_set_cancel_timeout() 101
rpc_mgmt_set_com_timeout() 102
rpc_mgmt_set_server_stack_size() 104
rpc_mgmt_stats_vector_free() 105
rpc_mgmt_stop_server_listening() 106
rpc_network_inq_protseqs() 107
rpc_network_is_protseq_valid() 108
rpc_ns_binding_export() 110
rpc_ns_binding_import_begin() 112
rpc_ns_binding_import_done() 114
rpc_ns_binding_import_next() 115
rpc_ns_binding_inq_entry_name() 118
rpc_ns_binding_lookup_begin() 120
rpc_ns_binding_lookup_done() 122
rpc_ns_binding_lookup_next() 123
rpc_ns_binding_select() 126
rpc_ns_binding_unexport() 127
rpc_ns_entry_expand_name() 129
rpc_ns_entry_inq_resolution() 130
rpc_ns_entry_object_inq_begin() 132
rpc_ns_entry_object_inq_done() 133
rpc_ns_entry_object_inq_next() 134

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_ns_group_delete() 136
rpc_ns_group_mbr_add() 137
rpc_ns_group_mbr_inq_begin() 139
rpc_ns_group_mbr_inq_done() 141
rpc_ns_group_mbr_inq_next() 142
rpc_ns_group_mbr_remove() 144
rpc_ns_mgmt_binding_unexport() 146
rpc_ns_mgmt_entry_create() 149
rpc_ns_mgmt_entry_delete() 150
rpc_ns_mgmt_entry_inq_if_ids() 151
rpc_ns_mgmt_handle_set_exp_age() 153
rpc_ns_mgmt_inq_exp_age() 154
rpc_ns_mgmt_set_exp_age() 156
rpc_ns_profile_delete() 158
rpc_ns_profile_elt_add() 159
rpc_ns_profile_elt_inq_begin() 161
rpc_ns_profile_elt_inq_done() 164
rpc_ns_profile_elt_inq_next() 165
rpc_ns_profile_elt_remove() 167
rpc_object_inq_type() 169
rpc_object_set_inq_fn() 171
rpc_object_set_type() 172
rpc_protseq_vector_free() 174
rpc_server_inq_bindings() 175
rpc_server_inq_if() 177
rpc_server_listen() 178
rpc_server_register_auth_info() 180
rpc_server_register_if() 182
rpc_server_unregister_if() 185
rpc_server_use_all_protseqs() 187
rpc_server_use_all_protseqs_if() 189
rpc_server_use_protseq() 191
rpc_server_use_protseq_ep() 193
rpc_server_use_protseq_if() 195
rpc_sm_allocate() 197
rpc_sm_client_free() 198
rpc_sm_destroy_client_context() 199
rpc_sm_enable_allocate() 200
rpc_sm_free() 201
rpc_sm_get_thread_handle() 202
rpc_sm_set_client_alloc_free() 203
rpc_sm_set_thread_handle() 204
rpc_sm_swap_client_alloc_free() 205
rpc_string_binding_compose() 206
rpc_string_binding_parse() 207
rpc_string_free() 209
uuid_compare() 211
uuid_create() 212
uuid_create_nil() 213
uuid_equal() 214
uuid_from_string() 215
uuid_is_nil() 216
uuid_to_string() 217

Section 4 Interface Definition Language **219**
4.1 Notation 219
4.2 IDL Language Specification 220
4.2.1 IDL Lexemes 220
4.2.1.1 Keywords and Reserved Words 220
4.2.1.2 Identifiers 220

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.1.3	IDL Punctuation	220
4.2.1.4	Alternate Representation of Braces	221
4.2.1.5	White Space	221
4.2.2	Comments	221
4.2.3	Interface Definition Structure	221
4.2.4	Interface Header	221
4.2.4.1	The uuid Attribute	222
4.2.4.2	The version Attribute	222
4.2.4.3	The endpoint Attribute	222
4.2.4.4	The local Attribute	222
4.2.4.5	The pointer_default Attribute	222
4.2.5	Interface Body	223
4.2.6	Import Declaration	223
4.2.7	Constant Declaration	223
4.2.7.1	Syntax	223
4.2.7.2	Semantics and Restrictions	224
4.2.8	Type Declarations and Tagged Declarations	224
4.2.9	Base Types	225
4.2.9.1	Syntax	225
4.2.9.2	Integer Types	225
4.2.9.3	The char Types	226
4.2.9.4	The boolean Type	226
4.2.9.5	The byte Type	226
4.2.9.6	The void Type	226
4.2.9.7	The handle_t Type	226
4.2.10	Constructed Types	226
4.2.11	Structures	226
4.2.12	Unions	226
4.2.12.1	Syntax	226
4.2.12.2	Semantics and Restrictions	227
4.2.13	Enumerated Types	228
4.2.14	Pipes	228
4.2.14.1	Syntax	228
4.2.14.2	Semantics and Restrictions	228
4.2.15	Arrays	228
4.2.15.1	Syntax	228
4.2.15.2	Semantics and Restrictions	229
4.2.15.3	Arrays of Arrays	229
4.2.16	Type Attributes	229
4.2.16.1	Syntax	229
4.2.16.2	Semantics and Restrictions	229
4.2.16.3	The transmit_as Attribute	229
4.2.16.4	The handle Attribute	230
4.2.16.5	The string Attribute	230
4.2.16.6	The context_handle Attribute	230
4.2.17	Field Attributes	230
4.2.17.1	Syntax	231
4.2.17.2	Semantics and Restrictions	231
4.2.17.3	The ignore Attribute	231
4.2.18	Field Attributes in Array Declarations	231
4.2.18.1	Conformant Arrays	231
4.2.18.2	Varying and Conformant Varying Arrays	232
4.2.18.3	Relationships Between Attributes	233
4.2.18.4	Negative Size and Length Specifications	233
4.2.19	Field Attributes in String Declarations	233
4.2.19.1	The first_is, last_is and length_is Attributes	233
4.2.19.2	The max_is Attribute	233
4.2.19.3	The size_is Attribute	234
4.2.20	Pointers	234

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

	4.2.20.1	Syntax	234
	4.2.20.2	Semantics and Restrictions	234
	4.2.20.3	Attributes Applicable to Pointers	235
	4.2.20.4	Varying Arrays of Pointers	235
	4.2.20.5	Restrictions on Pointers	236
	4.2.21	Pointers as Arrays	237
	4.2.21.1	Pointers with the string Attribute	237
	4.2.21.2	Possible Ambiguity Resolved	237
	4.2.22	Operations	237
	4.2.22.1	The idempotent Attribute	238
	4.2.22.2	The broadcast Attribute	238
	4.2.22.3	The maybe Attribute	238
	4.2.23	Parameter Declarations	238
	4.2.23.1	Syntax	238
	4.2.23.2	Semantics and Restrictions	238
	4.2.23.3	Directional Attributes	238
	4.2.23.4	Aliasing in Parameter Lists	239
	4.2.24	Predefined Types	239
	4.2.25	The error_status_t Type	239
	4.2.26	International Character Types	239
	4.2.27	Anonymous Types	240
	4.3	The Attribute Configuration Source	241
	4.3.1	Comments	241
	4.3.2	Identifiers	241
	4.3.3	Syntax	241
	4.3.4	Include Declaration	242
	4.3.5	Specifying Binding Handles	243
	4.3.5.1	The explicit_handle Attribute	243
	4.3.5.2	The implicit_handle Attribute	243
	4.3.5.3	The auto_handle Attribute	243
	4.3.6	The represent_as Attribute	244
	4.3.7	The code and nocode Attributes	244
	4.3.8	The in_line and out_of_line Attributes	244
	4.3.9	Return Statuses	245
	4.3.9.1	The comm_status Attribute	245
	4.3.9.2	The fault_status Attribute	245
	4.3.9.3	Interaction of the comm_status and fault_status Attributes	246
	4.3.10	The heap Attribute	246
	4.3.11	The enable_allocate Attribute	246
	4.4	IDL Grammar Synopsis	247
	4.4.1	Grammar Synopsis	247
	4.4.2	Alphabetic Listing of Productions	250
	4.5	IDL Constructed Identifiers	253
	4.6	IDL and ACS Reserved Words	254
Section	**5**	**Stubs**	**255**
	5.1	The Application/Stub Interface	255
	5.1.1	Parameters	255
	5.1.1.1	Parameter Memory Management	255
	5.1.1.2	Client-side Allocation	256
	5.1.1.3	Server-side Allocation	256
	5.1.1.4	Aliasing	257
	5.1.2	Default Manager EPVs	257
	5.1.3	Interface Handle	257
	5.1.4	Pipes	257
	5.1.4.1	Processing of in Pipes	259
	5.1.4.2	Processing of out Pipes	260
	5.1.4.3	Processing of in,out Pipes	261
	5.1.5	IDL and ACS Type Attributes	261
	5.1.5.1	The IDL transmit_as Attribute	261

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

	5.1.5.2	The IDL handle Attribute	262
	5.1.5.3	Interaction of IDL transmit_as and IDL handle Attributes	263
	5.1.5.4	The ACS represent_as Attribute	263
	5.1.5.5	Interaction of the ACS represent_as Attribute and the IDL handle Attribute	264
	5.1.5.6	Interaction of the ACS represent_as Attribute with the IDL transmit_as Attribute	264
	5.1.6	Context Handle Rundown	264
	5.2	Interoperability Requirements on Stubs	266
	5.2.1	Operation Numbers	266
	5.2.2	Error Handling During Floating-Point Unmarshalling	266
Section	**6**	**Remote Procedure Call Model**	**267**
	6.1	Client/Server Execution Model	268
	6.1.1	RPC Interface and RPC Object	268
	6.1.1.1	RPC Interfaces	268
	6.1.1.2	RPC Objects	268
	6.1.2	Interface Version Numbering	269
	6.1.2.1	Rules for Changing Version Numbers	269
	6.1.2.2	Definition of an Upwardly Compatible Change	269
	6.1.2.3	Non-upwardly Compatible Changes	269
	6.1.3	Remote Procedure Calls	269
	6.1.4	Nested RPCs	270
	6.1.5	Execution Semantics	270
	6.1.6	Context Handles	271
	6.1.7	Threads	271
	6.1.8	Cancels	273
	6.2	Binding, Addressing and Name Services	275
	6.2.1	Binding	275
	6.2.2	Endpoints and the Endpoint Mapper	276
	6.2.2.1	Client Operation	276
	6.2.2.2	Server Operation	277
	6.2.3	NSI Interface	277
	6.2.3.1	Common Declarations	277
	6.2.3.2	Protocol Towers	278
	6.2.3.3	The server_name Object Attributes	279
	6.2.3.4	The group Object Attributes	281
	6.2.3.5	The profile Object Attributes	282
	6.2.3.6	Encoding	283
	6.2.3.7	Name Service Class Values	283
	6.3	Error Handling Model	284
Section	**7**	**RPC Service Definition**	**285**
	7.1	Call Representation Data Structure	285
	7.2	Service Primitives	285
	7.2.1	Invoke	286
	7.2.2	Result	287
	7.2.3	Cancel	288
	7.2.4	Error	289
	7.2.5	Reject	290
Section	**8**	**Statechart Specification Language Semantics**	**291**
	8.1	The Elements of Statecharts	291
	8.2	State Hierarchies	293
	8.3	Concurrency	293
	8.4	Graphical Expressions	294
	8.4.1	Default Entrances	294
	8.4.2	Conditional Connectors	294
	8.4.3	Terminal Connectors	294
	8.5	Semantics that Require Special Consideration	295
	8.5.1	Implicit Exits and Entrances (Scope of Transitions)	295
	8.5.2	Conflicting Transitions	295

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

	8.5.3	Execution Steps and Time	295
	8.5.4	Synchronisation and Race Conditions	296
	8.6	Summary of Language Elements	297
	8.6.1	Event Expressions	297
	8.6.2	Condition Expressions	298
	8.6.3	Action Expressions	298
	8.6.4	Data Item Expressions	300
	8.6.4.1	Atomic Numeric Expressions	300
	8.6.4.2	Compound Numeric Expressions	300
	8.6.4.3	String Expressions	300
Section	**9**	**RPC Protocol Definitions**	**301**
	9.1	Conformance	301
	9.2	RPC Stub to Run-time Protocol Machine Interactions	302
	9.2.1	Client Protocol Machines	302
	9.2.2	Server Protocol Machines	303
	9.3	Connection-oriented Protocol	304
	9.3.1	Client/Server	304
	9.3.2	Association Group	304
	9.3.3	Association	304
	9.3.3.1	Association Management Policy	305
	9.3.3.2	Primary and Secondary Endpoint Addresses	305
	9.3.4	Call	305
	9.3.5	Transport Service Requirements	305
	9.4	Connection-oriented Protocol Machines	306
	9.4.1	CO_CLIENT_ALLOC	306
	9.4.2	CO_CLIENT_GROUP	306
	9.4.3	CO_CLIENT	306
	9.4.3.1	ASSOCIATION	307
	9.4.3.2	CONTROL	307
	9.4.3.3	CANCEL	307
	9.4.3.4	CALL	307
	9.4.4	CO_SERVER_GROUP	307
	9.4.5	CO_SERVER	307
	9.4.5.1	ASSOCIATION	308
	9.4.5.2	CONTROL	308
	9.4.5.3	CANCEL	308
	9.4.5.4	WORKING	308
	9.5	Connectionless Protocol	309
	9.5.1	Client/Server	309
	9.5.2	Activity	309
	9.5.3	Call	309
	9.5.4	Maintaining Execution Context and Monitoring Liveness	309
	9.5.5	Serial Numbers	310
	9.5.6	Transport Service Requirements	310
	9.6	Connectionless Protocol Machines	311
	9.6.1	RPC Stub to Run Time Protocol Machine Interactions	311
	9.6.2	CL_CLIENT	311
	9.6.2.1	CONTROL	311
	9.6.2.2	AUTHENTICATION	311
	9.6.2.3	CALLBACK	311
	9.6.2.4	PING	311
	9.6.2.5	CANCEL	311
	9.6.2.6	DATA	312
	9.6.3	CL_SERVER	312
	9.6.3.1	CONTROL	312
	9.6.3.2	AUTHENTICATION	312
	9.6.3.3	CANCEL	312
	9.6.3.4	WORKING	312
	9.7	Naming Conventions	313

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Section	**10**	**Connectionless RPC Protocol Machines**	**315**
	10.1	CL_CLIENT Machine	316
	10.1.1	CL_CLIENT Activities	316
	10.1.2	CL_CLIENT States	318
	10.1.3	CL_CLIENT Events	323
	10.1.4	CL_CLIENT Conditions	326
	10.1.5	CL_CLIENT Actions	331
	10.1.6	CL_CLIENT Data-Items	335
	10.2	CL_SERVER Machine	344
	10.2.1	CL_SERVER Activities	344
	10.2.2	CL_SERVER States	348
	10.2.3	CL_SERVER Events	354
	10.2.4	CL_SERVER Actions	358
	10.2.5	CL_SERVER Conditions	364
	10.2.6	CL_SERVER Data-Items	369
Section	**11**	**Connection-oriented RPC Protocol Machines**	**381**
	11.1	CO_CLIENT Machine	382
	11.1.1	CO_CLIENT Activities	382
	11.1.2	CO_CLIENT States	384
	11.1.3	CO_CLIENT Events	390
	11.1.4	CO_CLIENT Actions	396
	11.1.5	CO_CLIENT Conditions	400
	11.1.6	CO_CLIENT Data-Items	404
	11.2	CO_CLIENT_ALLOC Machine	413
	11.2.1	CO_CLIENT_ALLOC Activities	414
	11.2.2	CO_CLIENT_ALLOC States	415
	11.2.3	CO_CLIENT_ALLOC Events	417
	11.2.4	CO_CLIENT_ALLOC Actions	420
	11.2.5	CO_CLIENT_ALLOC Conditions	421
	11.2.6	CO_CLIENT_ALLOC Data-Items	422
	11.3	CO_CLIENT_GROUP Machine	423
	11.3.1	CO_CLIENT_GROUP States	424
	11.3.2	CO_CLIENT_GROUP Events	425
	11.3.3	CO_CLIENT_GROUP Data-Items	426
	11.4	CO_SERVER Machine	427
	11.4.1	CO_SERVER Activities	428
	11.4.2	CO_SERVER States	430
	11.4.3	CO_SERVER Events	435
	11.4.4	CO_SERVER Actions	441
	11.4.5	CO_SERVER Conditions	446
	11.4.6	CO_SERVER Data-Items	450
	11.5	CO_SERVER_GROUP Machine	459
	11.5.1	CO_SERVER_GROUP States	460
	11.5.2	CO_SERVER_GROUP Events	461
	11.5.3	CO_SERVER_GROUP Actions	463
	11.5.4	CO_SERVER_GROUP Data-Items	463
Section	**12**	**RPC PDU Encodings**	**465**
	12.1	Generic PDU Structure	465
	12.2	Encoding Conventions	466
	12.3	Alignment	466
	12.4	Common Fields	467
	12.4.1	PDU Types	467
	12.4.2	Protocol Version Numbers	467
	12.4.3	Data Representation Format Labels	467
	12.4.4	Reject Status Codes	467
	12.5	Connectionless RPC PDUs	468
	12.5.1	Connectionless PDU Structure	468
	12.5.2	Header Encoding	468
	12.5.2.1	Protocol Version Number	469

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.5.2.2	PDU Type	469
12.5.2.3	Flags Fields	469
12.5.2.4	Data Representation Format Label	470
12.5.2.5	Serial Number	470
12.5.2.6	Object Identifier	470
12.5.2.7	Interface Identifier	470
12.5.2.8	Activity Identifier	470
12.5.2.9	Server Boot Time	470
12.5.2.10	Interface Version	471
12.5.2.11	Sequence Number	471
12.5.2.12	Operation Number	471
12.5.2.13	Interface Hint	471
12.5.2.14	Activity Hint	471
12.5.2.15	PDU Body Length	471
12.5.2.16	Fragment Number	471
12.5.2.17	Authentication Protocol Identifier	472
12.5.3	Connectionless PDU Definitions	472
12.5.3.1	The ack PDU	472
12.5.3.2	The cancel_ack PDU	472
12.5.3.3	The cancel PDU	473
12.5.3.4	The fack PDU	473
12.5.3.5	The fault PDU	474
12.5.3.6	The nocall PDU	474
12.5.3.7	The ping PDU	475
12.5.3.8	The reject PDU	475
12.5.3.9	The request PDU	475
12.5.3.10	The response PDU	475
12.5.3.11	The working PDU	475
12.6	Connection-oriented RPC PDUs	476
12.6.1	Connection-oriented PDU Structure	476
12.6.2	Fragmentation and Reassembly	476
12.6.3	Connection-oriented PDU Data Types	477
12.6.3.1	Declarations	477
12.6.3.2	Connection-Oriented Protocol Versions	479
12.6.3.3	The frag_length Field	480
12.6.3.4	Context Identifiers	480
12.6.3.5	The call_id Field	480
12.6.3.6	The assoc_group_id Field	480
12.6.3.7	The alloc_hint Field	480
12.6.3.8	Authentication Data	480
12.6.3.9	Optional Connect Reject and Disconnect Data	481
12.6.4	Connection-oriented PDU Definitions	481
12.6.4.1	The alter_context PDU	481
12.6.4.2	The alter_context_resp PDU	483
12.6.4.3	The bind PDU	484
12.6.4.4	The bind_ack PDU	485
12.6.4.5	The bind_nak PDU	486
12.6.4.6	The cancel PDU	487
12.6.4.7	The fault PDU	488
12.6.4.8	The orphaned PDU	490
12.6.4.9	The request PDU	491
12.6.4.10	The response PDU	493
12.6.4.11	The shutdown PDU	494
Section 13	**Security**	**495**
13.1	The Generic RPC Security Model	496
13.1.1	Generic Operation	496
13.1.2	Generic Encodings	497
13.1.2.1	Protection Levels	497
13.1.2.2	Authentication Services	497

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

13.1.2.3	Authorisation Services	498
13.1.3	Underlying Security Services Required	498
13.2	Security Services for Connection-oriented Protocol	499
13.2.1	Client Association State Machine	499
13.2.2	Server Association State Machine	499
13.2.3	Sequence Numbers	499
13.2.4	The auth_context_id Field	500
13.2.5	Integrity Protection	500
13.2.6	Connection-oriented Encodings	501
13.2.6.1	Common Authentication Verifier Encodings	501
13.2.6.2	Encoding for Per-PDU Services	501
13.2.6.3	Credentials Encoding	503
13.3	Security Services for Connectionless Protocol	505
13.3.1	Server Receive Processing	505
13.3.2	Client Receive Processing	505
13.3.3	Conversation Manager Encodings	505
13.3.3.1	Challenge Request Data Encoding	505
13.3.3.2	Response Data Encoding	506
13.3.4	Authentication Verifier Encodings	506
13.3.4.1	dce_c_authn_level_none	506
13.3.4.2	dce_c_authn_level_connect	506
13.3.4.3	dce_c_authn_level_call	506
13.3.4.4	dce_c_authn_level_pkt	507
13.3.4.5	dce_c_authn_level_integrity	507
13.3.4.6	dce_c_authn_level_privacy	507
Section 14	**Transfer Syntax NDR**	**509**
14.1	Data Representation Format Label	510
14.2	NDR Primitive Types	511
14.2.1	Representation Conventions	511
14.2.2	Alignment of Primitive Types	511
14.2.3	Booleans	512
14.2.4	Characters	512
14.2.5	Integers and Enumerated Types	512
14.2.5.1	Enumerated Types	513
14.2.6	Floating-point Numbers	513
14.2.6.1	IEEE Format	514
14.2.6.2	VAX Format	515
14.2.6.3	Cray Format	516
14.2.6.4	IBM Format	517
14.2.7	Uninterpreted Octets	518
14.3	NDR Constructed Types	519
14.3.1	Representation Conventions	519
14.3.2	Arrays	519
14.3.2.1	Uni-dimensional Fixed Arrays	519
14.3.2.2	Uni-dimensional Conformant Arrays	520
14.3.2.3	Uni-dimensional Varying Arrays	520
14.3.2.4	Uni-dimensional Conformant-varying Arrays	521
14.3.2.5	Ordering of Elements in Multi-dimensional Arrays	521
14.3.2.6	Multi-dimensional Fixed Arrays	521
14.3.2.7	Multi-dimensional Conformant Arrays	521
14.3.2.8	Multi-dimensional Varying Arrays	522
14.3.2.9	Multi-dimensional Conformant and Varying Arrays	522
14.3.3	Strings	523
14.3.3.1	Varying Strings	523
14.3.3.2	Conformant and Varying Strings	524
14.3.4	Arrays of Strings	524
14.3.5	Structures	525
14.3.5.1	Alignment of Constructed Types	525
14.3.6	Structures Containing Arrays	526

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

	14.3.6.1	Structures Containing a Conformant Array	526
	14.3.6.2	Structures Containing a Conformant and Varying Array	527
	14.3.7	Unions	527
	14.3.8	Pipes	528
	14.3.9	Pointers	528
	14.3.10	Top-level Pointers	529
	14.3.10.1	Top-level Full Pointers	529
	14.3.10.2	Top-level Reference Pointers	529
	14.3.11	Embedded Pointers	530
	14.3.11.1	Embedded Full Pointers	530
	14.3.11.2	Embedded Reference Pointers	531
	14.3.11.3	Algorithm for Deferral of Referents	531
	14.4	NDR Input and Output Streams	532
Annex	**A**	**Universal Unique Identifier**	**533**
	A.1	Format	534
	A.2	Algorithms for Creating a UUID	536
	A.2.1	Clock Sequence	536
	A.2.2	System Reboot	536
	A.2.3	Clock Adjustment	537
	A.2.4	Clock Overrun	537
	A.2.5	UUID Generation	537
	A.3	String Representation of UUIDs	538
	A.4	Comparing UUIDs	539
Annex	**B**	**Protocol Sequence Strings**	**541**
Annex	**C**	**Name Syntax Constants**	**543**
Annex	**D**	**Authentication, Authorisation and Protection-level Arguments**	**545**
	D.1	The authn_svc Argument	545
	D.2	The authz_svc Argument	545
	D.3	The protect_level Argument	546
	D.4	The privs Argument	547
	D.5	The server_princ_name Argument	547
	D.6	The auth_identity Argument	547
	D.7	Key Functions	547
Annex	**E**	**Reject Status Codes and Parameters**	**549**
	E.1	Reject Status Codes	549
	E.2	Possible Failures	551
	E.2.1	comm_status Parameter	551
	E.2.2	fault_status Parameter	551
Annex	**F**	**IDL to C-language Mappings**	**553**
	F.1	Data Type Bindings	553
	F.2	Syntax Mappings	556
Annex	**G**	**Portable Character Set**	**559**
Annex	**H**	**Endpoint Mapper Well-known Ports**	**561**
Annex	**I**	**Protocol Identifiers**	**563**
Annex	**J**	**DCE CDS Attribute Names**	**565**
Annex	**K**	**Architected and Default Values for Protocol Machines**	**567**
Annex	**L**	**Protocol Tower Encoding**	**569**
	L.1	Protocol Tower Contents	570
Annex	**M**	**The dce_error_inq_text Manual Page**	**571**
		dce_error_inq_text()	572
Annex	**N**	**IDL Data Type Declarations**	**573**
	N.1	Basic Type Declarations	573
	N.2	Status Codes	575
	N.3	RPC-specific Data Types	577
Annex	**O**	**Endpoint Mapper Interface Definition**	**579**
Annex	**P**	**Conversation Manager Interface Definition**	**583**
	P.1	Server Interface	583

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

	P.2	Client Interface	585
Annex	**Q**	**Remote Management Interface**	**587**
Annex	**R**	**Mapping of RPC Protocol to an OSI Infrastructure**	**589**
	R.1	Abstract syntax name	589
	R.2	Transfer syntax name	589
	R.3	Simple RPC application context	590
	R.3.1	Application context name	590
	R.3.2	Component ASEs	590
	R.3.3	Mapping to supporting services	590

List of Figures

2-1	Information Required to Complete an RPC	20
2-2	Server Binding Relationships	24
2-3	Decisions in Looking Up an Endpoint	27
2-4	Decisions for Selecting a Manager	28
6-1	Execution Phases of an RPC Thread	272
6-2	Concurrent Call Threads Executing in Shared Execution Context	273
10-1	CL_CLIENT Statechart	316
10-2	CL_SERVER Statechart	344
11-1	CO_CLIENT Statechart	382
11-2	CO_CLIENT_ALLOC Statechart	413
11-3	CO_CLIENT_GROUP Statechart	423
11-4	CO_SERVER Statechart	427
11-5	CO_SERVER_GROUP Statechart	459
14-1	NDR Format Label	510
14-2	The Boolean Data Type	512
14-3	Character Data Type	512
14-4	NDR Integer Formats	513
14-5	IEEE Single-precision Floating-point Format	514
14-6	IEEE Double-precision Floating-point Format	515
14-7	VAX Single-precision (F) Floating-point Format	515
14-8	VAX Double-precision (G) Floating-point Format	516
14-9	Cray Floating-point Formats	517
14-10	IBM Floating-point Formats	517
14-11	Uninterpreted Octet Representation	518
14-12	Uni-dimensional Fixed Array Representation	520
14-13	Uni-dimensional Conformant Array Representation	520
14-14	Uni-dimensional Varying Array Representation	520
14-15	Uni-dimensional Conformant and Varying Array Representation	521
14-16	Multi-dimensional Fixed Array Representation	521
14-17	Multi-dimensional Conformant Array Representation	522
14-18	Multi-dimensional Varying Array Representation	522
14-19	Multi-dimensional Conformant and Varying Array Representation	523
14-20	Varying String Representation	523
14-21	Conformant and Varying String Representation	524
14-22	Multi-dimensional Conformant and Varying Array of Strings	525
14-23	Structure Representation	525
14-24	Representation of a Structure Containing a Conformant Array	526
14-25	Representation of a Structure Containing a Conformant and Varying Array	527
14-26	Union Representation	527
14-27	Pipe Representation	528
14-28	Top-level Full Pointer Representation	529
14-29	Top-level Reference Pointer Representation	530
14-30	Embedded Full Pointer Representations	531
14-31	Embedded Reference Pointer Representation	531
14-32	NDR Input Stream	532
14-33	NDR Output Stream	532

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

List of Tables

3-1	Client and Server Binding Handles	48
3-2	Rules for Returning an Object's Type	169
4-1	Integer Base Types	225
4-2	IDL Directional Attributes	238
4-3	Alphabetic Listing of Productions	250
4-4	Constructed Identifier Classes	253
5-1	Transmitted Type Routines	262
5-2	Transferred Type Routines	263
5-3	Floating Point Error Handling	266
6-1	Execution Semantics	270
6-2	Protocol Tower Structure	278
6-3	The server_name Object Attributes	279
6-4	RPC-specific Protocol Tower Layers	280
6-5	Example Protocol Tower	280
6-6	Service Group Object Attributes	281
6-7	Configuration Profile Object Attributes	282
7-1	Invoke Parameters	286
7-2	Result Parameters	287
7-3	Cancel Parameters	288
7-4	Error Parameters	289
7-5	Reject Parameters	290
8-1	Events Related to Other Elements	297
8-2	Compound Events	297
8-3	Conditions Related to Other Elements	298
8-4	Compound Conditions	298
8-5	Actions Related to Other Elements	298
8-6	Compound Actions	299
12-1	RPC Protocol Data Units	465
12-2	The First Set of PDU Flags	469
12-3	Second Set of PDU Flags	469
12-4	Authentication Protocol Identifiers	472
14-1	NDR Format Label Values	510
14-2	NDR Floating Point Types	513
A-1	UUID Format	534
A-2	UUID version Field	534
A-3	UUID variant Field	534
A-4	The 4 Msb of clock_seq_hi_and_reserved	537
A-5	Field Order and Type	539
B-1	RPC Protocol Sequence Strings	541
C-1	RPC Name Syntax Defined Constants	543
D-1	Casts for Authorisation Information	547
D-2	RPC Key Acquisition for Authentication Services	547
E-1	Reject Status Codes	549
E-2	Failures Returned in a comm_status Parameter	551
E-3	Failures Returned in a fault_status Parameter	551
F-1	IDL/NDR/C Type Mappings	553
F-2	Recommended Boolean Constant Values	554
G-1	Portable Character Set NDR Encodings	559
H-1	Endpoint Mapper Well-known Ports	561
I-1	NDR Transfer Syntax Identifier	563
I-2	Registered Single Octet Protocol Identifiers	563
J-1	DCE CDS Attribute Names	565
K-1	Default Protocol Machine Values	567
L-1	Floors 1 to 3 Inclusive	570
L-2	Floors 4 and 5 for TCP/IP Protocols	570
L-3	Floors 4, 5 and 6 for DECnet Protocol	570

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 11578 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 21, *Open systems interconnection, data management and open distributed processing*.

Annexes A to R form an integral part of this International Standard.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Information technology - Open Systems Interconnection -

Remote Procedure Call (RPC)

Section 1. General

Using RPC to develop distributed applications is growing in popularity and importance. This International Standard for OSI RPC will meet the growing demand for standardization in this area to ensure future interoperability of distributed applications making use of RPC.

This International Standard facilitates the specification and development of distributed applications based on extending the well-known remote procedure call paradigm to operate between application processes on two separate real open systems in the OSI environment.

The RPC issues addressed by this International Standard reflect the fact that RPC is related both to application programming environments and to communication systems. The modeling of RPC divides naturally into two complementary areas:

a) describing the RPC paradigm and how it relates to the application programming environments in which it is used, and

b) defining the communication service used to support RPC in an OSI environment.

This International Standard specifies both portability and interoperability for the Remote Procedure Call (RPC) mechanism. The specification contains material directed at two audiences:

- It provides a portability guide for application programmers.
- It provides both portability and interoperability specifications for those who are implementing or porting RPC or who are testing an RPC implementation.

This International Standard may be thought of as an implementation specification, covering both portability and interoperability, that contains within it an application portability guide. The application portability guide consists of Part 2, RPC Application Programmer's Interface and Part 3, Interface Definition Language and Stubs.

Although the portability specification is part of the broader implementation specification, it has been designed to stand alone so that it may be used by application programmers without reference to the other parts of the implementation specification.

Note: In order to make the portability specification independent, some material is repeated, especially between Section 2 and Section 6.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.1 Scope

This International Standard

a) defines the RPC Interaction Model for applications; it describes the styles of interactions supported by the OSI RPC Standard as perceived by application programmers;

b) defines the RPC Communication Model which describes the RPC components and characteristics within the framework of OSI Application Layer Structure;

c) defines the requirements met by the standard; and

d) defines the requirements to be met by other standards for use by RPC and defines the necessary components for the use of these standards.

e) specifies conformance requirements;

f) specifies the infrastructure of support environments for RPC such as naming, security and management;

g) specifies programming language bindings or specific programming language interfaces;

This International Standard

a) does not specify individual implementations or products;

b) does not specify local customizations and configurations;

c) does not specify the interface to local resources or access facilities that are provided within the local system; and

d) does not provide formal description tools to describe the algorithmic aspects of a distributed application or application semantics.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 7498-1:1994, *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*

ISO 7498-2:1989, *Information processing systems - Open Systems Interconnection - Basic Reference Model - Part 2: Security Architecture.*

ISO 7498-3:1989, *Information processing systems - Open Systems Interconnection - Basic Reference Model - Part 3: Naming and addressing.*

ISO 8649:1996, *Information technology - Open Systems Interconnection - Service definition for the Association Control Service Element.*

ISO 8822:1994, *Information technology - Open Systems Interconnection - Presentation service definition.*

ISO/IEC 8824:1990, *Information technology - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).*

ISO/IEC 9545:1994, *Information technology - Open Systems Interconnection - Application Layer Structure.*

ISO/IEC 10646-1:1993, *Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane.*

ISO/IEC 10731:1994, *Information technology - Open Systems Interconnection - Basic Reference Model - Conventions for the definition of OSI services.*

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.3 Informative References

X/Open P315: April 1996, *X/Open DCE: Authentication and Security Services.*

X/Open C312: December 1994, *X/Open DCE: Directory Services.*

IEEE P1003.4a:1995: *POSIX Threads.*

D. Harel, Science of Computer Programming, 8(3), 231-274, June 1987, *Statecharts: a visual formalism for complex systems.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1.4 Definitions

1.4.1 Reference model definitions

This International Standard is based on the concepts developed in ISO/IEC 7498, and makes use of the following terms defined in them:

a) Application Layer;

b) application-process;

c) application-entity;

d) application-service-element;

e) application-protocol-data-unit;

f) presentation-connection;

g) presentation context; and

h) real open system.

1.4.2 Application Layer Structure definitions

This International Standard makes use of the following terms defined in ISO 9545:

a) application-association;

b) application-entity-invocation;

c) application-process-invocation;

d) application-service-element-invocation;

e) application-service-object;

f) ASO-association;

g) ASO-context;

h) application-service-object-invocation; and

i) control function.

1.4.3 Service conventions definitions

This International Standard makes use of the following terms defined in ISO 10731:

a) OSI service;

b) service-provider;

c) service-user;

d) request;

e) indication;

f) response; and

g) confirm.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.5 Abbreviations

For the purposes of this International Standard, the following abbreviations apply:

ACL	Access Control List
ACS	Attribute Configuration Source
ACSE	Association Control Service Element
AND	Logical And operation
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASE	application-service-element
ASO	application-service-object
ASN.1	Asbtract Syntax Notation 1
BER	Basic Encoding Rules
BNF	Backus-Naur Form
CAS	Client Address Space
CDS	Cell Directory Service
CL	Connectionless
CLNS	Connectionless Network Service
CLTS	Connectionless Transport Service
CO	Connection-oriented
COTS	Connection Oriented Transport Service
CRC	Cyclic Redundancy Check
DDS	Domain Directory Service
DECNet	Digital Equipment Corporation Network products
DNA	Digital Network Architecture
DOD	US Department of Defense
EBCDIC	Extended Binary Coded Decimal Interchange Code
EPV	End Point Vector
GUID	Global User IDentifier
ID	Identifier
IDL	Interface Definition Language
IEEE	Institution of Electrical and Electronic Engineers
INF	Infinity
IP	Internet Protocol
IPX	Internetwork Packet Exchange
KB	Kilobyte
LHS	Left hand side
LSB	Least significant bit
MSB	Most significant bit
NaN	Not a number
NDR	Network Data Representation
NetBEUI	NetBIOS Extended User Interface
NetBIOS	Network Basic Input/Output System
NSAP	Network Service Access Point
NSI	Network Service Independent
NSP	Network Services Protocol
OID	Object Identifier
OR	Logical OR operation
OSI	Open Systems Interconnection
PAC	Privilege Attribute Certificate
PCS	Portable Character Set
PDU	Protocol Data Unit
POSIX	Portable Operating System
PSAP	Presentation Service Access Point
RHS	Right hand side
ROM	Read-only memory

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RPC	Remote Procedure Call
RTT	Round trip time
SPX	Sequenced Packet Exchange
TCP	Transmission Control Protocol
TCP/IP	TCP (q.v.)
TP4	OSI Transport Protocol Class 4
TPDU	Transport PDU (q.v.)
UDP/IP	UDP (q.v.)
UTC	Coordinated Universal Time
UUID	Unique Universal Identifier
XOR	Logical Exclusive OR operation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.6 Portability

The portability specification describes the concrete syntax and semantics of the Application Programmer's Interface (API) to RPC. It consists of:

- an introduction to the RPC API that describes the RPC programming model and gives general guidelines for portable usage (see Section 2)
- a reference subclause for the data types used in the RPC API (see Section 3)[1]
- a set of reference pages for the RPC run-time library routines; these specify the calling syntax and semantics for the interfaces (see Section 3)
- a reference to the Interface Description Language (IDL) (see Section 4)
- a mapping of IDL data types to ISO C data types (see Annex F)
- an RPC stub specification that defines stub characteristics required for portability (see Subclause 5.1 on page 255).

The portability specification is narrowly focussed on providing a guide to portable usage of the RPC API. It describes behaviour that is common to all implementations. Whenever implementation-specific behaviour is referenced, it is clearly marked as such. Similarly, the specification generally avoids examples or tutorial descriptions. Whenever usage guidelines are provided, they are clearly marked as such.

All behaviour that is not specifically marked as implementation-specific or a usage note, is considered to be required. All implementations must conform to the specified behaviour. Programmers can rely on the specified behaviour to be portable among conforming implementations.

1. This International Standard specifies ISO C-language bindings for data types and interfaces.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.7 Services and Protocols

The implementation specification includes a set of service and protocol specifications. The protocol specifications describe how implementations of the RPC client and server run-time systems communicate. The service specifications describe a set of abstract services that the RPC run-time system must implement.

The service and protocol specifications include:

- an abstract specification of the RPC model (see Section 6)
- an abstract specification of a set of RPC service primitives (see Section 7)
- abstract specifications of the RPC connectionless and connection-oriented communications protocols. These are given as sets of statecharts and associated descriptive materials. This includes an abstract specification of the underlying transport services required by the RPC protocols. (The protocol specifications are contained in Section 8, Section 9, Section 10 and Section 11.)
- byte stream specifications of the formats of RPC Protocol Data Units (PDUs) used by the connectionless and connection-oriented protocols (see Section 12) and common authentication verifier encodings (see Section 13)
- a specification of the Network Data Representation (NDR); this specifies a set of NDR data types and the byte stream formats in which they are communicated between client and server run-time environments (see Section 14)
- a mapping of IDL data types to NDR data types (see Annex F)
- an RPC stub specification that defines the stub characteristics required for interoperation (see Subclause 5.2 on page 266)
- a specification of information stored in and retrieved from name services (see Subclause 6.2 on page 275, Annex I and Annex J)
- a UUID specification (see Annex A)
- IDL data type declarations (see Annex N)
- the endpoint mapper protocol (see Annex O)
- the conversation manager protocol (see Annex P)
- the remote management interface (see Annex Q).

The aim of the service and protocol specifications is to provide a complete mapping from RPC call semantics to the byte streams that RPC run-time clients and servers interchange using underlying services. The RPC service primitives provide an abstract implementation of the specified RPC call semantics and serve to map the specified semantics to the specified protocol machines. The PDU formats give the byte streams that the protocol machines exchange using the underlying transport services. The NDR specification, along with the mapping of IDL to NDR data types, defines how the call data exchanged in the RPC PDUs is encoded.

Except for the byte stream specification and the stub specification, the service and protocol specifications are abstract. They describe the behaviour that conforming implementations must follow, but they do not prescribe any specific means for implementing this behaviour.

Implementations that conform to this specification interoperate according to the following rule: client and server applications, conforming to the same IDL source (but not necessarily the same ACS), correctly implement the specified RPC interface semantics for each remote procedure call operation specified in the IDL source.

Except when specified otherwise, IDL compiler behaviour and the stub, including the stub to run-time interface, are implementation-dependent. Therefore, the above rule applies when stubs are generated using the local implementation's IDL compiler. There is no requirement that stubs for a given language are portable among implementations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

1.8 Conformance Requirements

To conform to this International Standard, implementations must meet the following requirements:

- Implementations must support the endpoint selection rules in **Endpoint Selection** on page 27.
- Implementations must support the manager selection rules in **Interface and Manager Selection** on page 27.
- Implementations must support the search algorithm in Subclause 2.4.5.
- Implementations must support the API naming, syntax and semantics, as defined in Section 3. Implementations may extend the set of status codes International Standard.d in Section 3.
- Implementations must support the naming, syntax and semantics for IDL, as given in Section 4.
- Implementations must support the naming, syntax, and semantics for stubs, as given in Section 5.
- Implementations must support the semantics defined in Section 6.
- Implementations must support the NSI syntax and naming, as defined in Subclause 6.2 on page 275.
- Implementations must support the service semantics defined in Section 7.
- Implementations must follow the conformance rules specified in Section 9.
- Implementations must support the syntax of the PDU encodings in Section 12.
- Implementations must support the Authentication Verifier encodings, as defined in Section 13.
- Implementations must support the rules and encodings for NDR, as given in Section 14.
- Implementations must support the syntax, semantics and encoding for UUIDs, as defined in Annex A.
- Implementations must support the naming and semantics for protocol sequence strings, as defined in Annex B.
- Implementations must support the naming and semantics for the **name_syntax** arguments, as defined in Annex C.
- Implementations must support the naming and semantics for security parameters, as defined in Annex D.
- Implementations must support the naming and encodings for **comm_status** and **fault_status**, as defined Annex E.
- Implementations must support the mapping from IDL types to NDR types, and from NDR types to defined ISO C types, as defined in Annex F.
- Implementations must support the portable character set, as defined in Annex G.
- Implementations must use the endpoint mapper ports, as defined in Annex H for the corresponding protocols.
- Implementations must adhere to the rules for protocol identifier assignment, as defined in Annex I.
- Implementations must adhere to the mappings for Directory Service attributes, as defined in Annex J.
- Implementations must provide defaults for the protocol machine values specified in Annex K.
- Implementations must obey the special protocol tower encoding rules specified in Annex L.
- Implementations must support the syntax and semantics of the **dce_error_inq_text** routine specified in Annex M.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- Implementations must adhere to the mappings for transfer syntax UUIDs, as defined in Annex N.
- Implementations must support the endpoint mapper semantics, as defined in Annex O.
- Implementations must support the conversation manager semantics, as defined in Annex P.
- Implementations must support the remote management semantics as defined in Annex Q.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 2. Introduction to the RPC API

This Section provides a general description of the programming model implemented by the RPC Application Programming Interface (API). This description includes definitions of many of the concepts used throughout the RPC API manual pages. As such, it is a necessary prerequisite to the understanding of the manual pages, and the manual pages assume knowledge of this Section, even when they do not make explicit reference to it.

The description serves three purposes:

- It provides general information that is relevant to many of the routines in the RPC API, but is not specified in the individual manual pages.
- It provides a rationale for the set of RPC APIs included in this International Standard.
- It provides general guidelines for the intended use of the RPC APIs.

The general information covers topics, such as binding and name service usage, that are relevant to many of the manual pages. Typically, several routines perform tasks related to a given topic. This introduction provides a general model within which the tasks performed by individual routines and suites of routines can be understood. This general model also provides a rationale for the set of routines included in this International Standard. It describes the underlying operations required for RPC programming and shows how the set of RPC APIs included in this International Standard gives access to these operations.

In showing how the RPC API routines are meant to be used, this Section provides certain guidelines for consistent RPC client/server interface usage. These guidelines cover such areas as using the naming services and organising server resources. By following them, programmers can simplify the task of maintaining and enhancing server interfaces and writing client programs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.1 RPC Programming Model Overview

The RPC programming model can be viewed along two axes:

- client/server
- program/stub/run-time system.

Each view describes important aspects of the use of the RPC API.

2.1.1 Client/Server Model

The client/server view of RPC programming describes the distributed resource model implemented by the RPC mechanism. In this view, programming tasks are divided between servers, which provide services or make resources available to remote clients, and clients, which seek and make use of these services or resources.

2.1.1.1 Interfaces

The central component of the client/server model is the interface. An **interface** is a set of remotely callable **operations** offered by a server and invokable by clients. Interfaces are implemented by **managers**, which are sets of server routines that implement the interface operations. RPC offers an extensive set of facilities for defining, implementing and binding to interfaces.

The RPC mechanism itself imposes few restrictions on the organisation of operations into interfaces. RPC does provide a means to specify interface versions and a protocol to select a compatible interface version at bind time (see Section 4 and Section 6). When an interface is specified as a new version of an existing interface, the server manager code must provide the required version compatibility. Beyond this restriction, the programmer is free to place any set of remotely callable operations in a given interface.

2.1.1.2 Remoteness

The RPC paradigm makes remote calls an extension of the familiar local procedure call mechanism. Specifically, the call itself is made as a local procedure call, and the underlying RPC mechanism handles the remoteness transparently. Server interface programming is thus similar to local procedure call programming, except that the handler of the call runs in a separate address space and security domain.

From this point of view, a local procedure call is a special simple case of the more general call mechanism provided by RPC. RPC semantics extend local procedure call semantics in a variety of ways:

Reliability	Network transports may offer varying degrees of reliability. The RPC run-time system handles these transport semantics transparently, but RPC call specifications include a specification of **execution semantics** that indicates to the RPC protocols the required guarantees of success and the permissibility of multiple executions on a possibly unreliable transport. Server application code must be appropriate for the specified execution semantics.
Binding	RPC binding occurs at run time and is under program control. Client and server use of the RPC binding mechanism is discussed extensively in this Section.
No Shared Memory	Because calling and called procedures do not share the same address space, remote procedure calls with input/output parameters use copy-in, copy-out semantics. For the same reason, RPC has no notion of "global data structures" shared between the caller and callee; data must be passed via call parameters.
Failure Modes	A number of failure possibilities arise when the caller and callee are on physically separate machines. These include remote system or server crashes, communications failures, security problems and protocol incompatibilities. RPC includes a mechanism to return such remote errors to the caller.
Cancels	RPC extends the local cancel mechanism by forwarding cancels that occur during an RPC to the server handling the call, allowing the server application code to handle the cancel. RPC adds a cancel time-out mechanism to ensure that a caller can regain control within a specified amount of time if a cancelled call should fail to return.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Security — Executing procedures across physical machine boundaries and over a network creates additional requirements for security. The RPC API includes an interface to the underlying security services.

The RPC API provides programmers with the means to apply these extended semantics, but it shields applications from the rigours of transport level send-and-receive programming. The RPC programming paradigm gives the programmer control of the remote semantics at two points: in the interface specification and through the RPC API.

- The interface specification, while it is principally used to specify the local calling syntax of an interface, also allows programmers to specify the desired execution semantics, the degree to which binding is under program control and error semantics. Interface specification is described in Section 4.
- The RPC API gives applications access to a variety of run-time services and control of many client/server interactions at run time. Its most important function is to control the process of binding between clients and servers. Other functions include authentication, server concurrency and server management.

2.1.1.3 Binding

A remote procedure call requires a remote binding. The calling client must bind to a server that offers the interface it wants, and the client's local procedure call must invoke the correct manager operation on the bound-to server. Because the various parts of this process occur at run time, it becomes possible to exercise nearly total programmatic control of binding. The RPC API provides access to all aspects of the binding process.

Each binding consists a set of components that can be separately manipulated by applications, including protocol and addressing information, interface information and object information. This allows servers to establish many binding paths to their resources and allows clients to make binding choices based on all of the components. These capabilities are the basis for defining a variety of server resource models.

2.1.1.4 Name Services

Servers need to make their resources widely available, and clients need some way to find them without knowing the details of network configuration and server installation. Hence, the RPC mechanism supports the use of name services, where servers can advertise their bindings and clients can find them, based on appropriate search criteria. The RPC API provides clients and servers with a variety of routines that can be used to export and import bindings to and from name services.

2.1.1.5 Resource Models

The client/server model views servers as exporters of services — via RPC interfaces — and clients as importers of those services. Exported services typically take the form of access to resources, such as computational procedures, data, communications facilities, hardware facilities, or any other capabilities available to an application on a networked host. The RPC mechanism does not distinguish among such resource types in any way. On the contrary, it provides a uniform means of access — the remote procedure call — and allows the programmer to define the underlying resource model freely.

RPC does, however, provide specific mechanisms that implicitly support different approaches to resource modeling. These mechanisms take advantage of the flexibility of the binding process and the name services. The RPC mechanism supports three basic resource models:

By Server — In this model, clients seek to bind to a specific server instance that provides an interface of interest.

By Service — In this model, clients seek a service — as represented by an interface — without concern for the specific server instance that provides that service or any objects that the server manages.

By Object — In this model, clients seek a binding to any server that manages a specific object. An object may be any computational resource available to a server.

The RPC programming mechanism does not explicitly enforce these models. Instead, they are supported

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

implicitly by making available a set of run-time binding and name service facilities through the RPC API. Programmers may use these facilities according to their application requirements. However, this International Standard recommends that programs follow the models specified here in order to ensure consistent use of the client/server interface.

2.1.1.6 Security Services

The RPC API provides access to a variety of security services: client-to-server and server-to-client authentication, authorisation of access to server resources, and varying degrees of cryptographic protection of client/server communications.

2.1.1.7 Server Implementation

The client/server view of RPC is necessarily asymmetric. The model is based on providing services remotely via the export of RPC interfaces. Since servers are the means for implementing remote interfaces, the model is server-centred. The RPC architecture provides certain server facilities that make the implementation of servers more efficient. These include

Server Concurrency Implementations may buffer RPC requests at the server and automatically provide multiple threads to handle concurrent requests, relieving the application programmer of these tasks.

Remote Management The RPC run-time system automatically offers a set of remote server management interfaces that can be used for such purposes as querying and stopping servers.

2.1.2 Application/Stub/Run-time System

The application/stub/run-time system view of RPC describes the division of labour between application code and other RPC components in implementing a remote procedure call.

2.1.2.1 RPC Run Time

At the core of this model is the RPC run-time system, which is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.

The RPC API is the programmer's interface to the run-time system. The run time makes use of a number of services, such as the endpoint mapper, name services and security services. The RPC API also provides an interface to these services for carrying out RPC-specific operations. Portable usage of the RPC API is fully specified in this subclause of this International Standard.

2.1.2.2 Stubs

The stub is application-specific code, but it is not directly generated by the application writer and therefore appears as a separate layer from the programmer's point of view. The function of the stub is to provide transparency to the programmer-written application code. On the client side, the stub handles the interface between the client's local procedure call and the run-time system, marshaling and unmarshaling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps. On the server side, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

RPC transparency to the application programmer is provided by the interface specification mechanism. The programmer specifies interfaces using an Interface Definition Language (IDL), and the IDL compiler generates stubs automatically from the specification. Thus, the actual operations performed by the stub are largely invisible to the programmer, although they form part of the application-specific program code.

This Section does not cover the interface specification mechanism itself; this is International Standard.d in Section 4. What is covered here are the assumptions that the RPC programming model makes about stubs, such as well-known stub names and stub memory management.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.1.2.3 Application Code

RPC application code falls into two categories:

- remote procedure calls and manager code
- optional calls to the RPC API, mainly to set up the run-time system state required by remote procedure calls.

In the first category are the procedures written by the programmer to implement the client and server operations of the remote procedure call. On the client side, these are simply local calls to the stub interfaces for the remote procedures. On the server side, these are a set of manager routines that implement the operations of the interface. In most applications, manager routines are presumably a major part of the server code. Recall that, aside from requiring managers to conform to the specified execution semantics and version behaviour, the RPC mechanism imposes no specific constraints on manager implementations.

The programmer-written application code interacts with the RPC run-time system principally through the stub. This makes run-time operations largely transparent to the application code. Nevertheless, in order to control binding, security and other aspects of the RPC mechanism, the application often needs direct access to run-time operations. The RPC API provides applications with such access to the RPC run-time system and related services.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.2 API Operations

The RPC API provides access to an extensive set of run-time operations. Subclause 2.11 on page 40 provides a detailed taxonomy of APIs according to the operations performed. This subclause offers an overview, based on a somewhat broader set of categories.

- binding-related operations
- name service operations
- endpoint operations
- security operations
- stub memory management operations
- management operations
- UUID operations.

Subsequent subclauses of this Section cover many of these groups of operations in detail.

2.2.1 Binding-related Operations

Binding-related operations establish a relationship between a client and server that makes possible a remote procedure call. These operations may be roughly divided into two categories:

- operations to establish client/server communications using an appropriate protocol
- operations that establish internal call routing information for the server.

Operations in the first category include the creation of communications endpoints by the server for the set of protocols over which it wishes to receive remote procedure calls. Servers typically export information about the bindings thus created to a name service and an endpoint map. Clients typically import such binding information from a name service and an endpoint map (see Subclause 2.2.2 and Subclause 2.2.3).

Operations in the second category establish a set of mappings that the server can use to route calls internally to the appropriate manager routine. This routing is based on the interface and version, operation and any object requested by the call.

2.2.2 Name Service Operations

The RPC name service API includes an extensive set of operations for exporting and importing binding information to and from name services. These operations make use of a set of RPC-specific name service entry attributes to structure the exported binding information so that it can easily be found and interpreted by clients.

2.2.3 Endpoint Operations

Servers listen for remote procedure call requests over one or more protocol-specific endpoints. Typically, such endpoints are allocated dynamically when a server begins to listen, and their lifetime is only a single server instantiation. RPC provides an **endpoint mapper** mechanism that allows such volatile endpoint information to be maintained separately from the more stable components of a binding. Typically, servers export stable binding information to a name service and register their volatile endpoints with the local endpoint mapper. The endpoint mapper then resolves endpoints for calls made on bindings that do not contain them.

Endpoint operations are used by servers to register their endpoints with the endpoint mapper.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.2.4 Security Operations

These operations establish the authentication, authorisation services and protection levels used by remote procedure calls.

2.2.5 Stub Memory Management Operations

These operations are used by applications to manage stub memory. They are typically used by RPC applications that pass pointer data.

2.2.6 Management Operations

Management operations include a variety of operations with the potential to affect applications other than the one making the management call. Servers automatically export a set of remote management functions.

2.2.7 UUID Operations

UUIDs (Universal Unique Identifiers) are used frequently by the RPC mechanism for a variety of purposes. The UUID operations enable applications to manipulate UUIDs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.3 Binding

Binding refers to the establishment of a relationship between a client and a server that permits the client to make a remote procedure call to the server. In this International Standard, the term "binding" usually refers specifically to a protocol relationship between a client and either the server host or a specific endpoint on the server host, and "binding information" means the set of protocol and addressing information required to establish such a binding. But, for a remote procedure call, such a binding occurs in a context that involves other important elements, paralleling the notion of a binding in a local procedure call. In order for an RPC to occur, a relationship must be established that ties a specific procedure call on the client side with the manager code that it invokes on the server side. This requires both the binding information itself and a number of additional elements (see Figure 2-1). The complete list is as follows:

a) a protocol sequence that identifies the RPC and underlying transport protocols

b) an RPC protocol version identifier

c) a transfer syntax identifier

d) a server host network address

e) an endpoint of a server instance on the host

f) an object UUID that can optionally be used for selection among servers and/or manager routines

g) an interface UUID that identifies the interface to which the called routine belongs

h) an interface version number that defines compatibility between interface versions

i) an operation number that identifies a specific operation within the interface.

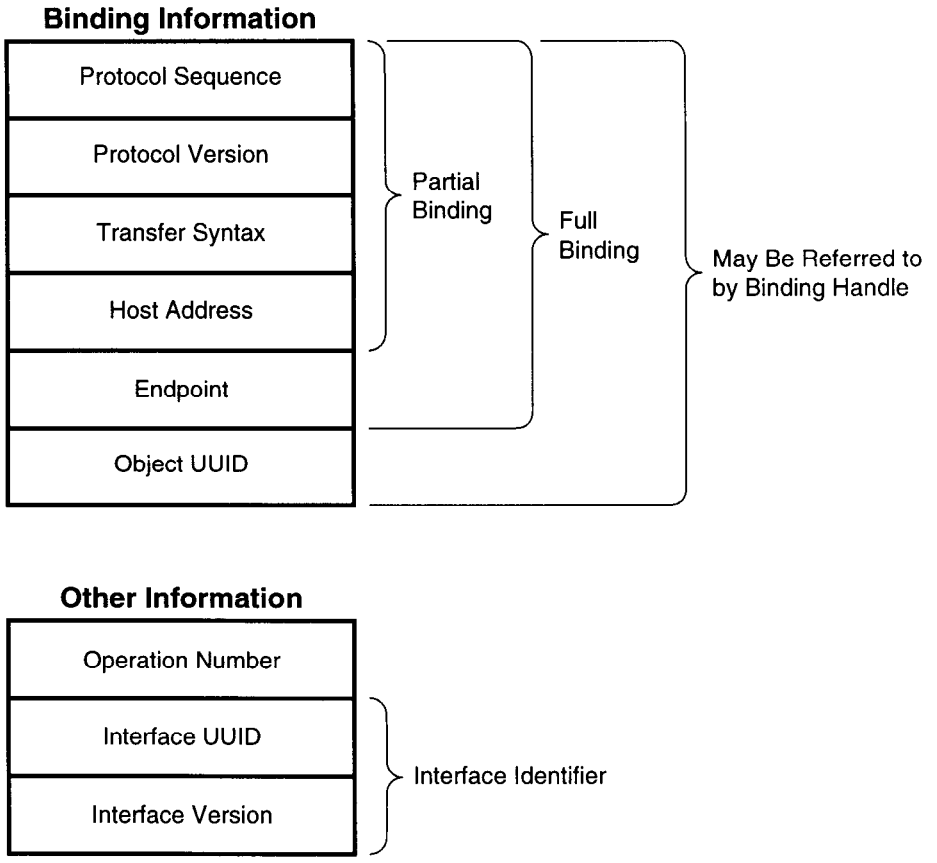


Figure 2-1 Information Required to Complete an RPC

Note: The discussion in this Section is intentionally vague about how any of this information is communicated between client and server. The underlying RPC protocol packages the required information for transmission. However, API usage is protocol-independent, and this Section provides a protocol-independent description of RPC. Hence, this Section typically refers to the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

binding information "contained" in a call without specifying how such information is actually transmitted or received. This is left to the RPC protocol specifications in Part 4, RPC Services and Protocols.

The binding information itself covers the first five elements of the list — the protocol and address information required for RPC communications to occur between a client and server.

Figure 2-1 on page 20 also shows the object UUID as part of the binding information. This is explained in Subclause 2.3.1.

In RPC terminology, such a binding can be partial or full. A **partial binding** is one that contains the first four elements of the list, but lacks an endpoint. A **full binding** contains an endpoint as well. The distinction is that a partial binding is sufficient to establish communications between a client and a server host, whereas a full binding allows communications to a specific endpoint on the server host.

2.3.1 Binding Handles

The binding information required to make remote procedure calls is maintained by the client and server run-time systems on behalf of applications. The run-time system provides applications with opaque binding handles to refer to locally maintained binding information. Applications use binding handles to manipulate bindings via calls to the RPC API.

It is important to understand that binding handles are only valid in the context of the local client or server instance that created them. They are not used directly to communicate binding information between servers and clients. Typically, servers advertise binding information by exporting it to name service entries. When a client imports binding information from a name service, it receives a binding handle from the client run-time system that refers to the local copy of the imported binding information.

Note: On the server side, such a binding handle refers to the first five elements shown in Figure 2-1 on page 20. On the client side, such a binding handle also refers to an object UUID associated with the binding information. For this reason, the figure includes the object UUID with the binding information even though it is not part of the protocol and address information required to establish communications between the client and server. The role of the object UUID is described in **Interface and Manager Selection** on page 27.

2.3.1.1 Client and Server Binding Handles

Binding information may refer either to a server or a client. Most of the time, binding information refers to servers, since it is servers to which clients need to bind in order to make remote procedure calls. When a binding refers to a server, a binding handle for it is called a **server binding handle**. Server binding handles are used **both by clients and servers** in the course of the binding process.

In some cases, servers need binding information for clients that call them. A binding handle that refers to such binding information is called a **client binding handle**. A small number of RPC APIs take client binding handles as arguments.

2.3.1.2 Obtaining Binding Handles

Applications obtain server binding handles by calling any of several RPC API routines. (See Subclause 3.1 on page 47 for a list of routines that return server binding handles.)

A server obtains a client binding handle as the first argument passed by the run-time system to a server manager routine.

2.3.2 String Bindings

A **string binding** is a string representation of binding information, including an optional object UUID. String bindings provide binding information in human-readable form. Applications can use RPC API calls to request a string binding from the run-time system or convert a string binding into a binding that the runtime system can use to make a remote procedure call. String binding format is specified in Subclause 3.1 on page 47.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.3.3 Binding Steps

In order to complete an RPC call, all of the elements listed in Figure 2-1 on page 20 must be present. RPC divides the process of assembling these elements into several steps and organises the assembled information in a way that provides maximum flexibility to the binding process. To understand this, consider the opposite possibility: a binding mechanism that seeks to imitate a local procedure call's static binding to a local library routine. In this case, all the elements would be preassembled into a well-known binding to which the calling program would bind in an all-or-nothing fashion.

RPC is close to the other dynamic extreme. It purposely avoids creating static links among all the elements so that a final routing — from the client procedure call to the server manager routine invoked — can be dynamically determined at the time of the RPC. From the programmer's point of view, one of the principal differences between a local procedure call and a remote procedure call is that the binding process — the way all these components are linked together — occurs at run time and can be carried out, optionally, under application program control.

This serves several purposes:

- It increases the location transparency of applications. Because clients do not need to know all the binding information before a call is actually made, applications can run successfully on systems with widely different configurations.
- It increases the maintainability of server installations because there are few **a priori** restrictions on the locations of server resources.
- It increases the probability of success in the face of partial failures because applications can look for bindings to servers in different locations and choose among a variety of RPC and network protocols.
- It makes possible a variety of server resource models by allowing servers to organise and advertise binding information in a variety of ways.

The binding process consists of a series of steps taken by the client and server to create, make available and assemble all the necessary information, followed by the actual RPC, which creates the final binding and routing using the elements established by the previous steps. To break the process down in more detail:

- The server takes a series of steps that establish binding-related state for the server side of the call.
- The server optionally exports binding information to a name service.
- The client takes a series of steps that establish binding-related state for the client side of the call. Binding information used in this process may be imported from a name service.
- The client makes a call, which is able to invoke the correct operation in the server by making use of the binding-related state established on the client and server sides.

Each of the components listed in Figure 2-1 on page 20 is involved at some stage of this process. Some components are involved at more than one stage and may be used in more than one way. The following subclauses consider each stage and component in some detail.

2.3.3.1 Server Binding Steps

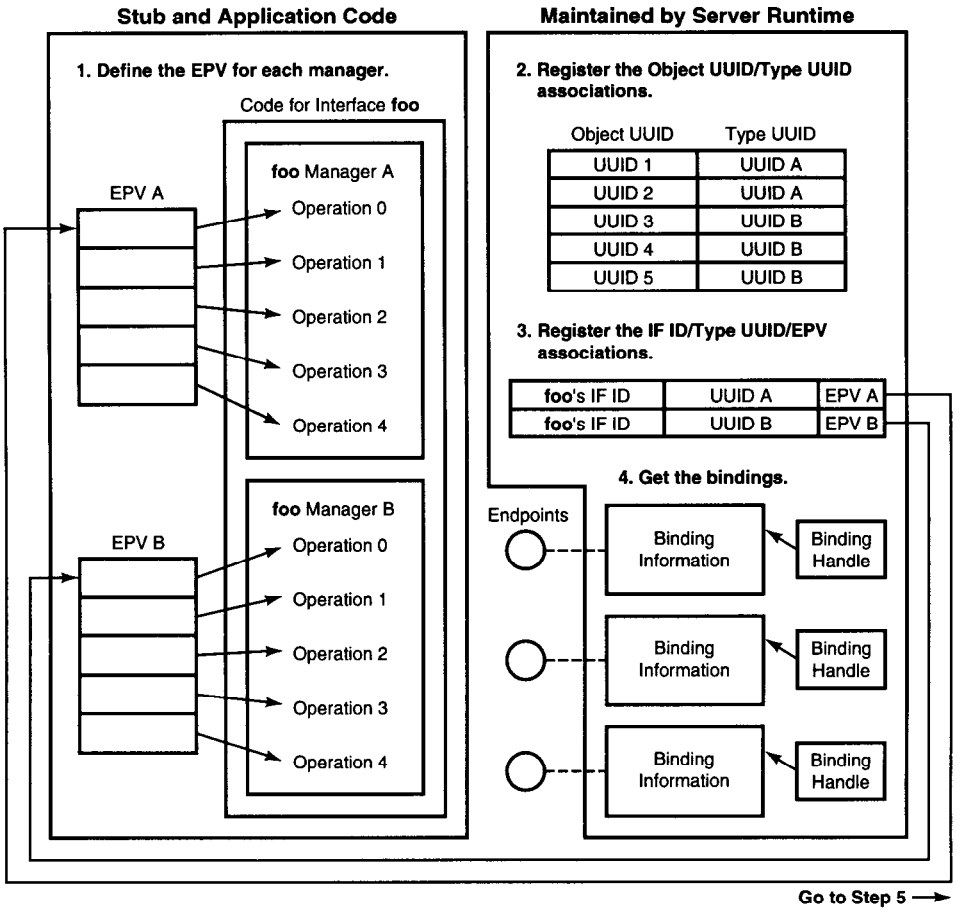
The server takes a number of steps to establish binding state in the server side run-time system, the name service and the endpoint mapper. The server's basic task is to acquire a set of endpoints from the run time and set up a series of relationships among binding elements that will then be used to construct the final routing at call time.

Figure 2-2 on page 24 shows the set of relationships that a server must establish to receive remote procedure calls. As the figure indicates, these are maintained in several places:

- by the server run-time system
- in the stub and application code

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- by the endpoint mapper
- by a name service.



Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

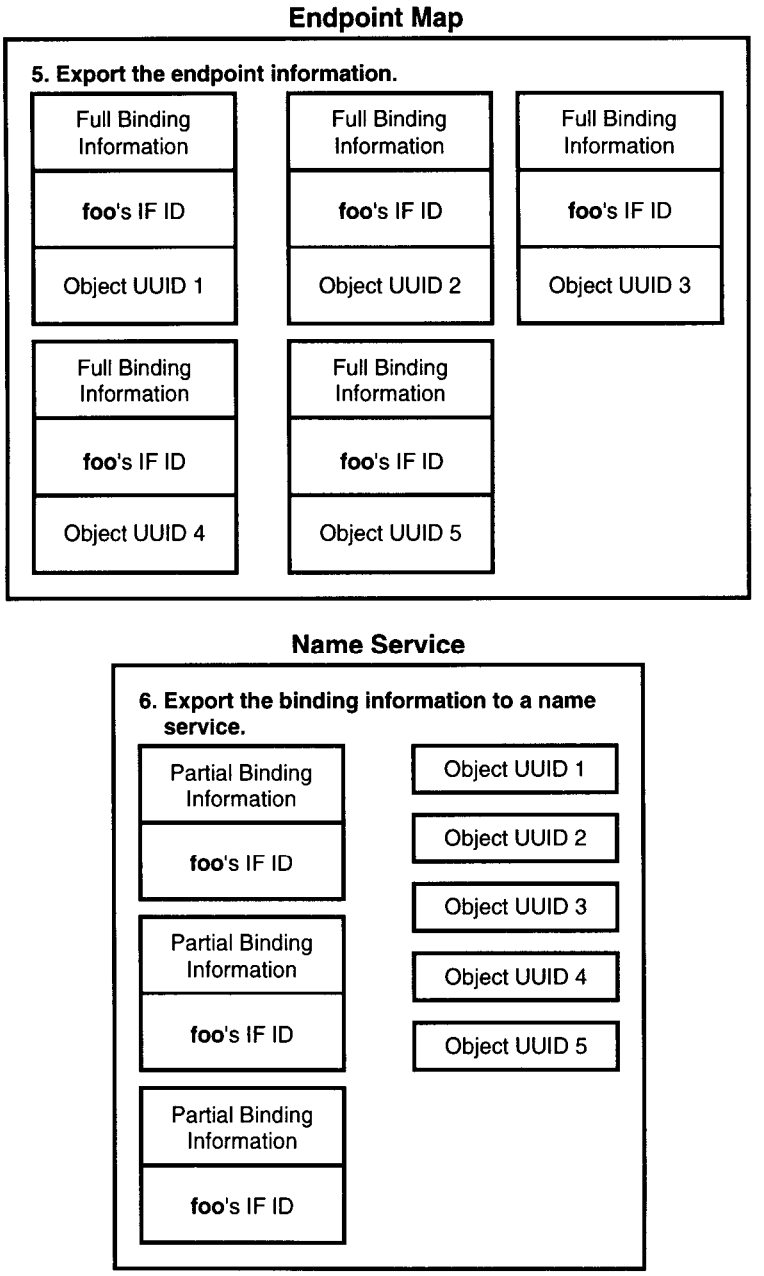


Figure 2-2 Server Binding Relationships

The server takes several steps (some of them optional) to establish the necessary relationships, as indicated in Figure 2-2. The steps are as follows:

a) The server application or stub code defines a manager Entry Point Vector (EPV) for each manager that the server implements. Recall that a manager is a set of routines that implements the operations of an interface. Recall also that servers may implement more than one manager for an interface; for example, to provide for different versions or object types. Each EPV is a vector of pointers to the operations of the manager. When an RPC request arrives, the operation number is used to select an element from one of the manager EPVs.

b) The server registers a set of object UUID/type UUID associations with the RPC run-time system.

c) The server registers interface identifier/type UUID/EPV associations with the RPC run-time system. Together with the previous step, this establishes the mappings that permit the run-time system to select the appropriate manager, based on the interface ID and any object UUID contained in a call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

d) The server application tells the run-time system what protocol sequences to use, and the run-time system establishes a set of endpoints for the protocol sequences requested. The server may ask the run-time system for its bindings, and the run time will return a set of binding handles that refer to the binding information for these endpoints.

e) The server may register binding information, consisting of a set of interface identifier/binding information/object UUID tuples, with the endpoint mapper. For each interface, the registered data consists of a cross product of the bindings and object UUIDs that the server wants to associate with that interface. When a call is received with a partial binding (that is, one lacking an endpoint) the endpoint mapper is able to use this information to select an endpoint that is capable of handling the call.

f) The server may export binding information to one or more name service entries. The information exported here looks quite similar to the information registered in the endpoint map in the previous step, with one important difference. The binding information exported to the name service generally lacks an endpoint, consisting of protocol and host address information only. Therefore the name service contains only the most persistent part of the binding information while the endpoint map contains the volatile endpoint portion.

   (The format is also different. See Subclause 2.4 on page 30 for information about the format of server entries.)

Note that not all of these steps are required. Servers may construct their own bindings, by using string bindings, rather than request them from the run-time system as described in step 4. Servers may also avoid exporting binding information to a name service and endpoint map as described in steps 5 and 6. In such a case, clients must then construct bindings from string bindings obtained by some other means.

Having completed the required steps, the server has established a set of relationships that allows the server run-time system to construct a complete binding, with routing to a specific server operation, for a call that contains the following information:

- full or partial binding information
- an interface identifier
- an object UUID, which may be nil
- an operation number.

The algorithms used are described in some detail in Subclause 2.4.5 on page 33. That discussion will show how the relationships established make possible a large number of paths to the interface and manager that are ultimately selected.

Note that the server run-time enironment itself maintains only a very limited set of relationships: interface identifier/type UUID/manager EPV and object UUIDs/type UUIDs. It is especially worth noting that the run-time system maintains no relationships between the protocol-address bindings it has created and any of the other information. The server merely advertises the relationships it wants to export in a name service and registers them in the endpoint map.

When the exported information is used by clients to find the server, client calls arriving at the server endpoints should contain interface identifier/object UUID pairs that the server can, in fact, service, although the RPC mechanism itself can provide no guarantee of this. This means that name service operations, while they are not, strictly speaking, a required part of an RPC call, usually play an important role in constructing bindings. Subclause 2.6 on page 36 shows how this makes the name service a key element in the organisation of server resources.

The indirect mapping from object UUID to type UUID to EPV (and hence to the manager called) also gives the server great flexibility in organising its resources based on objects UUIDs. This is explained in Subclause 2.6 on page 36.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.3.3.2 Client Binding Steps

The client binding steps are considerably simpler than those taken by the server. The basic task of the client is to find a suitable binding and use it to make a call, as described in the following steps.

Note: The following steps outline the **explicit binding method**. Client application code can avoid explicitly having to carry out step 1 by using the **automatic binding method**. In this case, the stub code takes care of importing suitable bindings. In step 2, clients can avoid having to supply an explicit binding handle for each call by choosing either the automatic or the implicit binding method. Binding methods are described in Subclause 2.3.3 on page 22 and Section 4.

a) Clients get suitable bindings by importing them from a name service. (Clients may also construct suitable bindings from binding information otherwise known to them, but here we describe the more general mechanism.)

   To make a call, the client needs a compatible binding: that is, one that offers the interface and version desired, uses a mutually supported protocol sequence, and if requested, is associated with a specific object UUID.

   Clients find compatible bindings by making calls to RPC API routines that search the name service. Recall that a name service entry binding attribute stores a set of associations between interface IDs and binding information. The client needs to find an element that specifies the desired interface and an acceptable protocol sequence and import the binding information from that element.

   Typically, the client specifies the interface desired, and the run-time system takes responsibility for finding bindings with protocol sequences that it can use. The client may also further select a specific protocol sequence.

   The client's selection of a binding may also depend on an object UUID. Recall that each name service entry may also store a set of object UUIDs. If the client requires a specific object UUID, it imports bindings only from name service entries that store that object UUID.

   For each binding that the client imports, the run-time system provides a server binding handle that refers to the binding information maintained by the client run-time system. This differs somewhat from the binding information referred to by a server binding handle on the server side. Recall that on the server, a server binding handle refers to a combination of protocol sequence and server address information. On the client side, a server binding handle may additionally refer to an object UUID, if the client has selected its bindings by object UUID.

b) Once the client has found a compatible binding, it makes a call using the binding handle for that binding. Depending on the binding method chosen, the client application code may supply the binding handle explicitly or it may leave this to the stub code (see Subclause 2.3.3 on page 22 and Section 4). When the call is made, the client run-time system has available to it the binding information and any object UUID referred to by the binding handle. Also available in the stub code are the interface identifier of the interface on which the call was made, and the operation number of the routine being called. Recall that the last three items of this tuple of information — the object UUID/interface identifier/operation number — are precisely what the server needs to route the call to a specific manager operation.

2.3.3.3 Call Routing Algorithms

Once the server and client have taken all the necessary steps to set up server and client side relationships, the call mechanism can use them to construct a complete binding and call routing when the call is made. This subclause specifies the algorithms used. In following these algorithms, it may be useful to refer to Figure 2-2 on page 24 to see how each of the relationships described there is used.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Endpoint Selection

When the client makes a call with a binding that lacks an endpoint, the endpoint is acquired from the endpoint mapper on the target host. The endpoint mapper finds a suitable endpoint by searching the local endpoint map for a binding that provides the requested interface UUID, and if requested, object UUID. The flowchart in Figure 2-3 shows the algorithm.

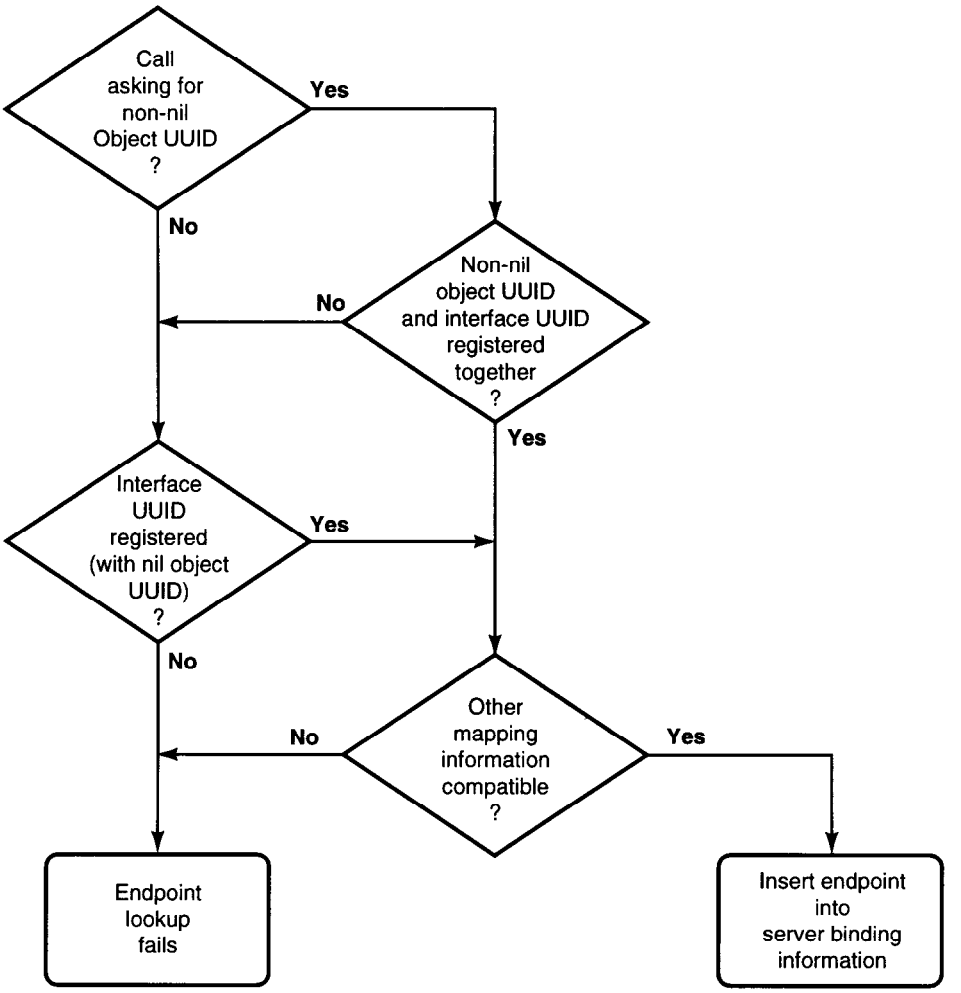


Figure 2-3 Decisions in Looking Up an Endpoint

What is important to note in this algorithm is that the interface and protocol information must match to find an endpoint, but an object UUID match may not be required. A server can provide a default UUID match by registering the nil UUID. Calls with a nil or unmatched object UUID follow the default path.

The endpoint map permits multiple endpoints to be registered with identical interface, protocol and object UUID information. Such endpoints are assumed to be interchangeable, and the endpoint mapper selects among them using an implementation-dependent algorithm.

Interface and Manager Selection

Having selected an endpoint, a call can be routed to one of the endpoints being used by a compatible server instance. The server can unambiguously select the correct interface and operation by using the interface identifier and operation number contained in the call. A call's interface identifier matches an interface identifier registered by the server when the interface UUIDs and major version numbers are equal and the call's minor version number is less than or equal to the minor version number registered by the server.

Recall, however, that the RPC mechanism makes it possible for a server to implement multiple managers for an interface. Hence it may be necessary to select the correct manager. Manager selection is based on the object UUID contained in the call. The selection mechanism depends on two of the relationships established

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

by the server: the object UUID/type UUID mapping and the interface ID/type UUID/manager UUID mapping. The flowchart in Figure 2-4 shows the selection algorithm.

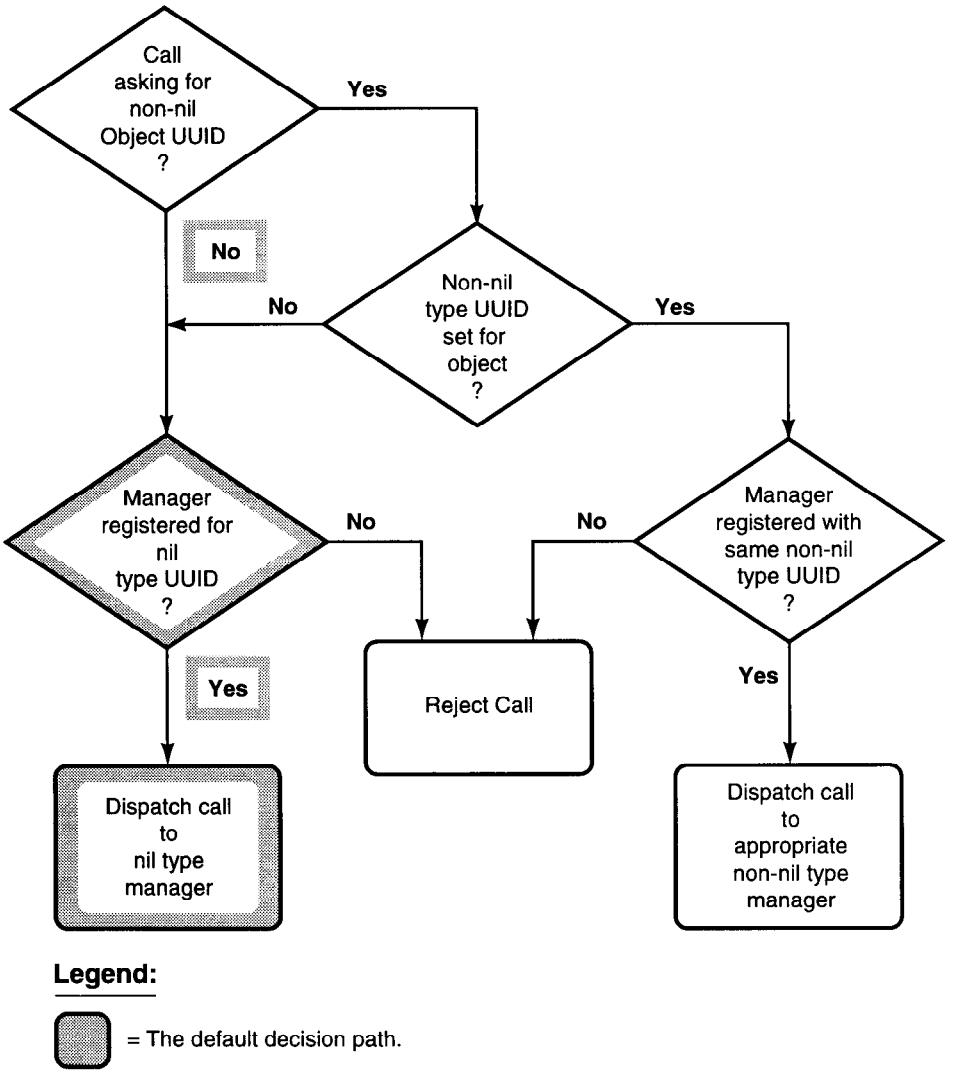


Figure 2-4 Decisions for Selecting a Manager

Here the server provides a default path by registering a default manager for the nil type UUID. Calls containing the nil object UUID, or any UUID for which the server has not set another type UUID, will be directed to the default manager.

Dispatching via the Manager EPV

Once the manager is selected, the call is dispatched via the selected manager EPV. Recall that a manager EPV is a vector of pointers to manager routines, one for each operation of the interface. The operation number is used to select the appropriate routine.

The actual call — via the manager EPV — to the server manager code is made by the server stub. Up to this point, the binding discussion has deliberately avoided questions of implementation. The run-time system maintains a set of relationships logically required by the binding algorithms, but the way in which these are implemented is entirely outside the purview of this International Standard. The case of the manager EPV is different, however. The manager EPV is an interface-specific data structure that must be declared by server code. The stub normally declares a default manager EPV, but when there is more than one manager for an interface, the application code must declare further manager EPVs. Subclause 3.1 on page 47 shows how to construct the appropriate declaration.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.3.4 Binding Methods

Client applications can exercise varying degrees of control over the binding process outlined in Subclause 2.3.3.2 on page 26.

- Using the explicit binding method, the client specifies a binding handle as an explicit parameter of each RPC. With this method, the client may choose a specific binding as often as once per call. The client carries out step 1, as described in Subclause 2.3.3.2 on page 26, as often as necessary to create the bindings it requires.
- Using implicit binding, the client specifies a binding handle globally for an interface, and the client stub automatically supplies the global binding for each call made on the interface. Using this method, the client needs to carry out step 1 only once per interface.
- Using automatic binding, the client allows the stub to import suitable bindings for it automatically. Using this method, the client does not carry out step 1, and does not supply a binding handle when making a call.

The automatic and implicit binding methods are interface wide and thus mutually exclusive. The explicit binding method may be specified per call and takes precedence over implicit or automatic binding specified for an interface.

Clients applications choose a binding method for an interface by specifying an ACS binding attribute, as International Standard.d in Section 4.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.4 Name Service Interface

The RPC API provides an extensive name service interface that applications use to export and import binding information. In general, name services can support much broader usage, but the RPC API is designed to support the RPC binding mechanism, rather than as a generalised name service interface. The following subclauses describe those aspects of name services that are relevant to the name service interface and binding.

The name service interface is designed to be independent of the underlying name service. Hence, it is referred to as the **Name Service-independent (NSI) interface**. As far as possible, these subclauses describe the name service interface without reference to any specific underlying name service. However, applications using the name service interface need to pass name service-specific names to the interface and therefore must be aware of the details of naming for the underlying services. These issues are discussed in Subclause 2.4.2.

2.4.1 Name Service Model

The name service interface is designed to allow servers to export binding information, and clients to find it, in an efficient manner. The interface permits servers to organise their binding information in a variety of ways. These support the server resource models described in Subclause 2.6 on page 36.

The name service interface makes two general assumptions about the underlying name service:

- The name service maintains a namespace database, the entries of which are accessible via names with some name service-specific syntax.
- The name service leaf entries can support a set of RPC-specific attributes that the name service interface uses when it exports, searches for and imports binding information.

The name service interface is used to store associations between bindings, interfaces and objects in name service entries. For each interface offered by a server, the server exports a set of protocol towers to the name service. A **protocol tower** combines binding information (not including an object UUID) for a single binding with an interface identifier. The set of protocol towers exported for an interface thus represents available bindings to the server for that interface. Servers can also export sets of object UUIDs associated with arbitrary resources they offer. The binding information exported by servers may be organised in a number of name service entries. The API makes use of several entry attributes, as described in Subclause 2.4.3 on page 31, to store binding-related information.

Clients make name service API calls to search for suitable bindings, specifying the interface and, possibly, any object UUID they are interested in, as well as a starting point for the search. The name service search operations search name service entries and return bindings that are compatible with the requirements of the client.

A client search of the namespace beginning at a given entry follows a path through name service entries determined by the algorithm given in Subclause 2.4.5 on page 33. The name service interface permits applications to define prioritised paths through the namespace, including default paths. Default paths make it possible to minimise the amount of knowledge about the namespace required by a client to begin searching for bindings.

2.4.2 Name Syntax Tags

The name service interface maintains its name service independence by using **name syntax tags**. Each interface that takes an entry name argument also takes an entry name syntax tag argument that indicates which name service syntax is to be used to interpret the name. Supported values for this argument are specified in Annex C.

RPC ISO C implementations provide an **RPC_DEFAULT_ENTRY_SYNTAX** environment variable that specifies a default entry name syntax tag.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.4.3 Name Service Attributes

The name service interface defines four RPC-specific name service attributes. These are as follows:

Binding Attribute — The binding attribute stores a set of protocol towers. An entry with the binding attribute is known as a **server entry**.

Group Attribute — The group attribute stores a set of entry names of the members of a single group. An entry with the group attribute is known as a **group entry**.

Profile Attribute — The profile attribute stores a set of profile elements. An entry with the profile attribute is known as a **profile entry**.

Object Attribute — The object attribute stores a set of object UUIDs.

While the name service interface does not impose any explicit restrictions on the use of these entries (there are no enforced schema), the name service model is designed to support applications that structure their name service entries according to the following recommended rules:

- Applications should create distinct binding, group and profile entries. While any name service entry can contain any combination of the four name service entry attributes, applications should not place binding, group and profile attributes in the same entry. The object attribute should appear only in server and group entries.
- Each server entry must contain information about only one server instance.
- Each group entry should contain information about only one interface and its versions, or one object, or one set of interchangeable server instances.

The following subclauses describe the contents of the entry types in detail.

2.4.3.1 Server Entries

Server entries contain bindings for a single server. Server entries may also contain an object attribute that specifies a set of object UUIDs associated with the server.

The binding attribute in a server entry stores a set of protocol towers. Recall that a protocol tower consists of an interface identifier along with binding information. Typically, the binding information lacks an endpoint so that the information represents a partial binding.

The information stored by the binding attribute does not include object UUIDs. Instead, when a server wishes to associate object UUIDs with the bindings stored in a server entry, it exports them to an object attribute in that entry. As described in **Interface and Manager Selection** on page 27, object UUIDs may be used to map calls to object-specific type managers, but servers may also use object UUIDs to identify any arbitrary server resource. When clients import bindings, they can specify object UUIDs so as to import bindings only for servers that provide a required resource. This usage of object UUIDs plays an important role in the server resource models described in Subclause 2.6 on page 36.

2.4.3.2 Group Entries

A group entry contains names of one or more server entries, other groups or both. A group provides a way to organise the server entries of different servers that offer a common RPC interface or object. Since a group can contain group names, groups can be nested. Each server entry or group named in a group is a **member** of the group. A group's members should offer one or more RPC interfaces, objects or both in common.

2.4.3.3 Profiles

A profile is an entry that contains a prioritised set of profile elements. A **profile element** is a database record that corresponds to a single RPC interface and that refers to a server entry, group or profile. Each profile element contains the following information:

- Interface Identifier

  This field is the search key for the profile. The interface identifier consists of the interface UUID and the interface version numbers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- Member Name

  The entry name of one of the following kinds of name service entries:

  — a server entry for a server offering the requested RPC interface

  — a group corresponding to the requested RPC interface

  — a profile.

- Priority Value

  The priority value is used by NSI operations to determine the order in which elements are searched. The search algorithm described in Subclause 2.4.5 on page 33 specifies how these values are used. Priority values range from 0, which is the highest priority, to 7, which is the lowest.

- Annotation String

  The annotation string is textual information used to identify the profile. It is not used by NSI search operations but can provide valuable information to namespace and server administrators.

Additionally, a profile can contain at most one **default profile element**. A default profile element is the element that a name service search operation uses when a search using the other elements of a profile finds no compatible binding information. A **default profile** is a profile referenced by a default profile element. Default profiles are typically used as an administrative device to optimise clients' searches for compatible bindings.

2.4.4 Binding Searches

Routines to extract information from a name service are present in the API in suites of three. Each suite includes:

- a **begin** routine
- a **next** routine
- a **done** routine.

In general, applications use these suites as follows:

a) The application obtains a **name service handle** by calling the **begin** routine. RPC Name Service routines use name service handles to refer to search state information maintained by the run-time system. The data type declaration for these handles is described in Subclause 3.1 on page 47.

b) The application calls the **next** routine one or more times using the handle obtained in step 1. Each call returns another element, or set of elements, along the path being searched.

c) The application calls the **done** routine using the handle obtained in step 1 to terminate the search.

The **begin** routine returns a handle used by a subsequent series of search operations. The handle refers to information maintained by the run-time system about the search, including search context information — such as matching criteria — and information about the current state of the search. Each call to the **begin** routine returns a handle that maintains the context for a distinct series of subsequent search operations.

The **next** routine returns elements, or sets of elements, one by one along the path being searched. The application calls this routine one or more times with a handle obtained from the **begin** routine. Each call returns another element or a status code that indicates that no more elements remain. Calls to the **next** routine using the same handle form part of one series of search operations along a search path. Calls to the **next** routine using different handles pertain to distinct and independent searches.

The **done** routine frees the search context referred to by the handle and invalidates the handle.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.4.5 Search Algorithm

The name service search operations traverse a path through one or more entries in the name service database when searching for compatible binding information. The path taken by any name service search, beginning at a given entry, depends on the organisation of binding information using the various name service entry attributes. This subclause describes the algorithm used by name service searches to determine what steps to take at each traversed entry.

In each name service entry, searches ignore non-RPC attributes and process the name service entry attributes in the following order:

a) the binding attribute (and object attribute, if present)

b) the group attribute

c) the profile attribute.

If a search path includes a group attribute, the search path can encompass every entry named as a group member. If a search path includes a profile attribute, the search path can encompass every entry named as the member of a profile element that contains the target interface identifier.

The following pseudocode presents the algorithm for retrieving bindings from a namespace. This describes the order in which bindings are returned by the routines **rpc_ns_binding_import_done**() and **rpc_ns_binding_lookup_next**().

In the pseudocode, each `entryName`, group member and profile element represent names that may be found in the namespace. Associated with each of these entries in the namespace may be any of the eight possible combinations of the **binding**, **group** and/or **profile** attributes.

The order in which bindings are returned is significant and is indicated in the algorithm. This algorithm only indicates the order of search. Local buffering constraints may cause the search to be interrupted and resumed.

```
Procedure GetBindings (someName) {
  /* "someName" represents the name of an entry in the namespace. */

  /* The following procedure recursively searches for bindings */
  Procedure Search(entryName)
    {
      Check entryName for binding attribute;
      If (binding attribute found)
        {
          Retrieve bindings from binding attribute;
          Randomise the bindings obtained from this attribute;
          Add these bindings to the bottom of the global list of bindings;
        }

      Check entryName for group attribute;
      If (group attribute found)
        {
          Retrieve members from group attribute and save in a list;
          Randomise the members in this list;
          Do
            {
              Select the first member and remove from the list;
              /* */
              /* Cycle checking requires knowledge of other   */
              /* names referenced within the scope of a call  */
              /* to GetBindings.                              */
              /* */
              Check for a cycle;
              If (not a cycle)
                {
                  If (member selected exists)
                    {
                      Search (member selected);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
                    }
                }
            }
          Until (list of members is empty);
        }

      Check entryName for profile attribute;
      If (profile attribute found)
        {
          Retrieve elements from profile attribute and save in a list;
          Sort profile elements in list by priority, highest first;
          Randomise the profile elements within each priority;
          Do
            {
              Select the first profile element and remove from the list;
              /* */
              /* Cycle checking requires knowledge of other    */
              /* names referenced within the scope of a call   */
              /* to GetBindings.                               */
              /* */
              Check for a cycle;
              If (not a cycle)
                {
                  If (element selected exists)
                    {
                      Search (element selected);
                    }
                }
            }
          Until (list of profile elements is empty);
        }
    }

  /* This is the body of the main routine starting the search */

  Initialize a global ordered list of bindings to empty;
  Search (someName);
  return ordered list of bindings;
}
```

2.4.6 Name Service Caching

Name service interface operations may cache name service data to avoid unnecessary lookups in the name service database. Whether caching occurs is implementation-dependent, but it is expected that most implementations will use caching. For implementations that cache, this International Standard specifies the semantics of caching to be governed by an **expiration age** as follows. Cached name service data is given an expiration age when it is cached. Name service interface operations use the cached copy when it has not outlived its expiration age. When a name service interface operation refers to cached data that has outlived its expiration age, the data is looked up in the name service database and the cache is updated.

The RPC run-time system sets the expiration age to a default value. Applications can specify another value either globally for the application or for a specific name service handle. The global value applies, by default, to all name service operations performed by the application. A handle-specific value applies only to operations performed using a specific name service handle.

When an application changes its global expiration age, or even the expiration age for a single handle, the effects may not be entirely confined to the application itself. Frequent updates of name service cache data may affect the performance of other clients of the name service and applications sharing the same cache. For this reason, operations that affect expiration age are considered to be management operations.

A non-caching implementation may be considered as a degenerate case of a caching implementation that behaves as if every cache item had outlived its expiration age.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.5 Server Model

The RPC model is server-centred in the sense that RPC provides many facilities to support varied and powerful server implementations, often with relatively little programming effort. These include:

- support for multiple interfaces, versions, objects and managers, as described in Subclause 2.3 on page 20
- automatic server concurrency and request buffering
- support for remote management.

2.5.1 Server Concurrency and Request Buffering

The RPC design assumes that servers export resources that may be widely available and possibly in high demand. The RPC model therefore provides for automatic concurrent service and buffering of RPC requests.

RPC provides server concurrency without requiring application code to spawn additional threads or processes explicitly. When beginning to listen for a call, the server application requests a number of call threads, and the RPC run-time system automatically provides the requested threads, up to an implementation-defined limit. Applications that request more than one call thread must, however, implement manager routines in a thread-safe manner.

Implementations may also allow additional requests that cannot be executed concurrently to be queued for subsequent execution. Otherwise they are rejected. Applications may make buffer size requests when registering a protocol sequence, although the actual buffer size provided is implementation-dependent.

2.5.2 Management Interface

Servers automatically implement, in addition to the interfaces specified by the programmer, a set of remote management interfaces that can be used for such operations as making remote inquiries to and stopping servers. These are accessible, both locally and remotely, via management RPC routines.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.6 Server Resource Models

The RPC API gives programs a high degree of control of the process by which bindings are constructed, component by component. This allows programs to specify the precise service required by any given instance of a remote procedure call. At the same time, the name service interface permits applications to structure binding information stored by a name service in a variety of ways. Together, these capabilities are the basis for a variety of strategies for organising server resources, based on the way the components of a binding are made available by a server.

The RPC API does not require server resources to be organised in any specific way; it simply provides facilities that permit a variety of forms of organisation. The resource models outlined here are only conventions. However, this International Standard recommends following these conventions. Servers provide resources that may be widely available, and they make use of a common resource — the name services — to advertise their bindings. Organising server resources according to well-defined conventions makes it easier to construct clients that can find the resources they need.

This International Standard recommends three basic server resource models:

- the server-oriented model
- the service-oriented model
- the object-oriented model.

These models are not mutually exclusive.

2.6.1 The Server-Oriented Model

In the server-oriented model, it is the server that is of interest to clients looking for bindings. In the simplest case, each server exports its bindings to one server entry and clients can go directly to a server entry to find bindings. Server instances may be interchangeable if they are running on the same host and offer the same interfaces and objects. Entries for interchangeable server instances may be organised as a group, and clients may begin their binding searches at the group entry.

2.6.2 The Service-Oriented Model

In the service-oriented model, clients are interested in some service, as defined by an interface (and its versions). The interface may be exported by more than one server, and server entries for servers that export a given interface may be organised in the same group. However, client applications seeking services normally do not have knowledge of the local namespace that will lead them directly to the required group entry. Typically, such clients use profiles to find the local instantiations of services they want.

2.6.3 The Object-Oriented Model

In the object-oriented model, a server associates some resource that it offers with an object UUID. Several servers may offer the same interface but different objects. Each server then exports the object UUIDs it offers to one or more separate server entries.

In order to make object UUIDs available to clients seeking a specific object, servers offering an object typically export object UUIDs to a group entry for that object. The group entry name is thus effectively associated with the object. Clients seeking a specific object can begin by importing an object UUID from the group entry for the object. The client then imports bindings for the object and interface it wants, beginning its search with the object entry.

Servers that export object UUIDs may or may not explicitly map these to type managers. In the simplest case, the server only registers an interface with a nil type UUID, causing all calls on the interface to be handled by the default manager. In this case, the association between object UUID and resource exists only in the namespace, and the server must assume that a client interested in a given object has, in fact, imported its binding correctly. On the other hand, servers may use object/type mappings to dispatch calls precisely according to object UUID. (See Subclause 2.3.3 on page 22 for the details of the mappings and selection algorithm.)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.7 Security

The RPC API provides a small number of interfaces that applications can use to set the authentication and authorisation services and the protection levels used by remote procedure calls. Servers that want to use authenticated RPC register a set of **server principal name/authentication service** pairs with the run-time system. To make an authenticated call, a client associates security information with a binding on which it is going to call, including a server principal name and authentication, authorisation and protection-level information.

Once the required authentication state is set, authentication and protection are carried out transparently by the RPC run-time system, using the specified services. If the server principal name and authentication service specified by the client do not match a pair registered by the server, the call fails. A server can specify a non-default authentication key retrieval function, but is not otherwise required (or allowed) to implement any of the authentication mechanism.

If the authentication requested is successful, the server manager routine can retrieve the caller's authentication, authorisation and protection-level information from the run-time system. Since the server may have registered more than one principal name/authentication service pair, the application code may still want to make an authentication decision at this point.

The server manager code also makes authorisation decisions based on the authorisation information it retrieves from the run-time system. The server is free to use this authorisation information to make whatever authorisation decisions are appropriate for the application.

The RPC security-related API is designed to be independent of any specific authentication and authorisation services. Servers and clients specify the required services via parameters to the authentication-related calls. The run-time system carries out authentication using the requested authentication service, passes authorisation service-specific authorisation information with the call, and provides protection that corresponds (in a service specific way) to the requested protection level. Supported values for the authorisation, authentication and protection-level parameters are specified in Annex D.

2.8 Error Handling

The RPC API provides a consistent error handling mechanism for all routines. Each routine includes a **status** output argument, which is used to return error status codes. These codes may be passed to the **dce_error_inq_text**( ) routine to extract error message text from a message catalogue. (See **dce_error_inq_text**( ) on page 572.)

RPC calls return protocol and run-time error status codes through **fault_status** and **comm_status** parameters, as described in Section 4. These status codes are consistent with the status codes returned from the RPC API and may be passed to **dce_error_inq_text**( ) to obtain error message text.

The status codes International Standard.d in this International Standard must be supported by all implementations. Implementations may support additional status codes, but these are not required.

2.9 Cancel Notification

RPC provides a remote cancel notification mechanism that can forward asynchronous cancel notifications to servers. When a client thread receives a cancel notification during an RPC, the run-time environment forwards the notification to the server. When the server run-time system receives the forwarded notification, it attempts to notify the server application thread that is handling the call. This can result in one of three outcomes for the RPC call on the client side:

a) If the notification is delivered to and handled by the server application thread, the RPC returns normally to the client.

b) If the server run-time system is unable to deliver the notification to the server application thread (for example, because the server application is blocking notifications), the notification is returned to the client run-time system. The RPC returns normally to the client, and the client run-time system attempts to deliver the notification to the client application thread. The client application code may

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

then handle the notification.

c) If the notification is delivered to the server application thread, but the server application code fails to handle it, the RPC returns to the client with a fault status.

Client applications may want to avoid waiting an indeterminate amount of time before a cancelled call returns. The RPC mechanism therefore allows client applications to specify a cancel time-out period. If a cancel occurs during an RPC, and the cancel time-out period expires before the call returns, the call returns to the client with a fault status. Such a call is said to be orphaned at the server. An orphaned call may continue to execute in the server, but it cannot return to the client.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2.10 Stubs

While stubs are generally transparent to the application code, applications may need to be aware of certain stub characteristics:

- IDL to stub data type mappings
- manager EPVs
- interface handles
- stub memory management.

This version of this International Standard specifies C-language stub bindings only.

2.10.1 IDL to Stub Data Type Mappings

Stubs generated from the IDL specification of an interface contain language-specific bindings for the interface operations. Client calls to remote procedures, and the server operations that implement these procedures, must conform to the bindings defined by the stubs. Therefore, applications must be aware of the mappings from the IDL data types that appear in an interface specification to the data types that appear in the stub declarations.

The C-language mappings are specified in Annex F. As specified there, stubs use defined types rather than primitive C-language types in declarations. Applications should use these defined types to ensure that their type declarations are consistent with those of the stubs, even when the application is ported to a different platform.

2.10.2 Manager EPVs

Stubs may contain a default manager EPV as described in Subclause 3.1 on page 47 Applications that declare additional nondefault manager EPVs must avoid the default name.

2.10.3 Interface Handles

Each stub declares an interface handle, which is a reference to interface specific information that is required by certain RPC APIs. (See Subclause 3.1 on page 47 for an explanation of how applications can access the declared interface handle.)

2.10.4 Stub Memory Management

RPC attempts to extend local procedure call parameter memory management semantics to a situation in which the calling and called procedure no longer share the same memory space. In effect, parameter memory has to be allocated twice, once on the client side, once on the server side. Stubs do as much of the extra allocation work as possible so that the complexities of parameter allocation are transparent to applications. In some cases, however, applications may have to manage parameter memory in a way that differs from the usual local procedure call semantics. This typically occurs in applications that pass pointer parameters that change value during the course of the call. Detailed rules for stub memory management by applications are given in Section 5 and Subclause 5.1.1.1 on page 255.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.11 RPC API Routine Taxonomy

The following subclauses summarise the RPC API routines, classifying them according to the kinds of functions they perform.

Note: Implementations of the RPC API must be synchronous cancel-safe (in the context of POSIX threads). Implementations of the RPC API need not be asynchronous cancel-safe. Multi-threaded implementations must be thread-safe.

2.11.1 Binding Operations

The routines in this group manipulate binding information. Most of these routines use binding handle parameters to refer to the underlying binding information. The string binding routines provide a way to manipulate binding information directly in string format.

A number of routines from the Object Operations and the Authentication and Authorisation groups also manipulate the information referenced by binding handles.

rpc_binding_copy() Returns a binding handle that references a new copy of binding information.

rpc_binding_free() Releases a binding handle and referenced binding information resources.

rpc_binding_from_string_binding() Returns a binding handle from a string representation of a binding handle.

rpc_binding_reset() Resets a server binding so the host remains specified, but the server instance on that host is unspecified.

rpc_binding_server_from_client() Converts a client binding handle to a server binding handle.

rpc_binding_to_string_binding() Returns a string representation of a binding handle.

rpc_binding_vector_free() Frees the memory used to store a vector of binding handles and the referenced binding information.

rpc_server_inq_bindings() Returns binding handles for RPC communications.

rpc_string_binding_compose() Combines the components of a string binding into a string binding.

rpc_string_binding_parse() Returns, as separate strings, the components of a string binding.

2.11.2 Interface Operations

The routines in this group manipulate interface information. Many of these routines take interface handle arguments. These handles are declared by stubs to reference the stubs' interface specifications. The routine **rpc_server_register_if**() is used to establish a server's mapping of interface identifiers, type UUIDs and manager EPVs. The routine **rpc_if_inq_id**() can be used to return the interface identifier (interface UUID and version numbers) from an interface specification.

rpc_if_id_vector_free() Frees the memory used to store a vector and the interface identifier structures it contains.

rpc_if_inq_id() Returns the interface identifier for an interface specification.

rpc_server_inq_if() Returns the manager entry point vector registered for an interface.

rpc_server_register_if() Registers an interface with the RPC run-time system.

rpc_server_unregister_if() Unregisters an interface from the RPC run-time system.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.11.3 Protocol Sequence Operations

The routines in this group deal with protocol sequences. The various **server_use*** routines are used by servers to tell the run-time system which protocol sequences to use to receive remote procedure calls. After calling one of these routines, the server calls **rpc_server_inq_bindings**() to get binding handles for all the protocol sequences on which it is listening for calls.

rpc_network_inq_protseqs()	Returns all protocol sequences supported by both the RPC run-time system and the operating system.
rpc_network_is_protseq_valid()	Tells whether the specified protocol sequence is valid and supported by both the RPC run-time system and the operating system.
rpc_protseq_vector_free()	Frees the memory used by a vector and its protocol sequences.
rpc_server_use_all_protseqs()	Tells the RPC run-time system to use all supported protocol sequences for receiving remote procedure calls.
rpc_server_use_all_protseqs()	Tells the RPC run-time system to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls.
rpc_server_use_protseq()	Tells the RPC run-time system to use the specified protocol sequence for receiving remote procedure calls.
rpc_server_use_protseq_ep()	Tells the RPC run-time system to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls.
rpc_server_use_protseq_if()	Tells the RPC run-time system to use the specified protocol sequence combined with the endpoints in the specified interface specification for receiving remote procedure calls.

2.11.4 Local Endpoint Operations

The routines in this group manipulate information in an application host's local endpoint map. These include the routines that servers typically use to register and unregister their binding information in the local endpoint map. A set of endpoint management routines is also available for more general manipulation of local and remote endpoint maps.

rpc_ep_register()	Adds to, or replaces, server address information in the local endpoint map.
rpc_ep_register_no_replace()	Adds to server address information in the local endpoint map.
rpc_ep_resolve_binding()	Resolves a partially bound server binding handle into a fully bound server binding handle.
rpc_ep_unregister()	Removes server address information from the local endpoint map.

2.11.5 Object Operations

The routines in this group manipulate object related information. Servers use **rpc_object_set_type**() to establish their object UUID/type UUID mappings. Clients typically specify the object UUID they wish to associate with a binding when they import bindings from a name service. However, clients can use **rpc_binding_set_object**() to associate a different object UUID with a binding. Servers can use **rpc_object_set_inq_fn**() to establish private object UUID/type UUID mappings.

rpc_object_inq_type()	Returns the type of an object.
rpc_object_set_inq_fn()	Registers an object inquiry function.
rpc_object_set_type()	Assigns the type of an object.
rpc_binding_inq_object()	Returns the object UUID from a binding handle.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_binding_set_object() Sets the object UUID value into a server binding handle.

2.11.6 Name Service Interface Operations

The routines of this group constitute most of the RPC name service independent interface (NSI). A group of name service management routines is also available. The NSI routines are divided into several subcategories according to groups of functions.

2.11.6.1 NSI Binding Operations

Applications use the routines in this subgroup to the export and import bindings to and from name service server entries. These include two suites of **begin/next/done** routines that applications can use to import bindings.

rpc_ns_binding_export() Exports server binding information to a name service entry.

rpc_ns_binding_import_begin() Creates an import context for importing bindings from a name service.

rpc_ns_binding_import_done() Deletes a name service import context.

rpc_ns_binding_import_done() Returns a binding handle for a compatible server from a name service.

rpc_ns_binding_inq_entry_name() Returns the name of an entry in the name service database from which the binding information referenced by a server binding handle came.

rpc_ns_binding_lookup_begin() Creates a lookup context for importing bindings from a name service.

rpc_ns_binding_lookup_done() Deletes a name service lookup context.

rpc_ns_binding_lookup_next() Returns a vector of binding handles for compatible bindings from a name service.

rpc_ns_binding_select() Returns a binding handle from a vector of compatible server binding handles.

rpc_ns_binding_unexport() Removes binding information from an entry in a name service database.

2.11.6.2 NSI Entry Operations

Applications use the routines in this group to return information about name service entries of various types.

rpc_ns_entry_expand_name() Expands the name of a name service entry.

rpc_ns_entry_object_inq_begin() Creates an inquiry context for viewing the objects stored in an entry in a name service database.

rpc_ns_entry_object_inq_done() Deletes a name service object inquiry context.

rpc_ns_entry_object_inq_next() Returns an object stored in an entry in a name service database.

2.11.6.3 NSI Group Operations

Applications use the routines in this group to manipulate name service group entries.

rpc_ns_group_delete() Deletes a group attribute.

rpc_ns_group_mbr_add() Adds an entry name to a group; if necessary, creates the entry.

rpc_ns_group_mbr_inq_begin() Creates an inquiry context for viewing group members.

rpc_ns_group_mbr_inq_done() Deletes the inquiry context for a group.

rpc_ns_group_mbr_inq_next() Returns a member name from a group.

rpc_ns_group_mbr_remove() Removes an entry name from a group.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.11.6.4 NSI Profile Operations

Applications use the routines in this group to manipulate name service profile entries.

rpc_ns_profile_delete( )	Deletes a profile attribute.
rpc_ns_profile_elt_add( )	Adds an element to a profile; if necessary, creates the entry.
rpc_ns_profile_elt_inq_begin( )	Creates an inquiry context for viewing the elements in a profile.
rpc_ns_profile_elt_inq_done( )	Deletes the inquiry context for a profile.
rpc_ns_profile_elt_inq_next( )	Returns an element from a profile.
rpc_ns_profile_elt_remove( )	Removes an element from a profile.

2.11.7 Authentication Operations

Applications use the routines in this group to manipulate the authentication, authorisation and protection-level information used by authenticated remote procedure calls.

rpc_binding_inq_auth_client( )	Returns authentication information referenced by a client binding handle.
rpc_binding_inq_auth_info( )	Returns authentication information referenced by a server binding handle.
rpc_binding_set_auth_info( )	Sets authentication information referenced by a server binding handle.
rpc_server_register_auth_info( )	Registers authentication information with the RPC run-time system.

2.11.8 The Server Listen Operation

This routine performs the final step in server initialisation, causing the server to begin to listen for remote procedure calls.

rpc_server_listen( )	Tells the RPC run-time system to listen for remote procedure calls.

2.11.9 The String Free Operation

Applications use this routine to free the string memory allocated by RPC API routines that return strings.

rpc_string_free( )	Frees a character string allocated by the run-time system.

2.11.10 UUID Operations

The routines in this group manipulate UUIDs.

uuid_compare	Compares two UUIDs and determines their order.
uuid_create	Creates a new UUID.
uuid_create_nil	Creates a nil UUID.
uuid_equal	Determines if two UUIDs are equal.
uuid_from_string	Converts a string UUID to binary representation.
uuid_hash	Creates a hash value for a UUID.
uuid_is_nil	Determines if a UUID is nil.
uuid_to_string	Converts a UUID from binary representation to a string representation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2.11.11 Stub Memory Management

The routines in this group enable applications to participate in stub memory management.

rpc_sm_allocate() Allocates memory within the RPC stub memory management scheme.

rpc_sm_client_free() Frees memory allocated by the current memory allocation and freeing mechanism used by the client stubs.

rpc_sm_destroy_client_context() Reclaims the client memory resources for a context handle, and sets the context handle to NULL.

rpc_sm_disable_allocate() Releases resources and allocated memory within the RPC stub memory management scheme.

rpc_sm_enable_allocate() Enables the stub memory management environment.

rpc_sm_free() Frees memory allocated by the **rpc_sm_allocate**() routine.

rpc_sm_get_thread_handle() Gets a thread handle for the stub memory management environment.

rpc_sm_set_client_alloc_free() Sets the memory allocation and freeing mechanism used by the client stubs.

rpc_sm_set_thread_handle() Sets a thread handle for the stub memory management environment.

rpc_sm_swap_client_alloc_free() Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client.

2.11.12 Endpoint Management Operations

The routines in this group provide a more general interface for manipulating endpoint maps than the one provided by the Local Endpoint Operations group. Routines in this group allow the examination of endpoint map elements one at a time and permit operations both on the application host's local endpoint map and on remote endpoint maps. These are considered management operations because of their potential to affect applications other than the one making the management call.

rpc_mgmt_ep_elt_inq_begin() Creates an inquiry context for viewing the elements in a local or remote endpoint map.

rpc_mgmt_ep_elt_inq_done() Deletes the inquiry context for viewing the elements in a local or remote endpoint map.

rpc_mgmt_ep_elt_inq_next() Returns one element at a time from a local or remote endpoint map.

rpc_mgmt_ep_unregister() Removes server address information from a local or remote endpoint map.

2.11.13 Name Service Management Operations

The routines in this group carry out operations typically done by name service management applications or only infrequently done by most applications. These are considered management operations because of their potential to affect applications other than the one making the management call.

rpc_ns_mgmt_binding_unexport() Removes multiple binding handles, or object UUIDs, from an entry in a name service database.

rpc_ns_mgmt_entry_create() Creates an entry in a name service database.

rpc_ns_mgmt_entry_delete() Deletes an entry from a name service database.

rpc_ns_mgmt_entry_inq_if_ids() Returns the list of interfaces exported to an entry in a name service database.

rpc_ns_mgmt_handle_set_exp_age() Sets a handle's expiration age for cached copies of name service data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_ns_mgmt_inq_exp_age() Returns an application's global expiration age for cached copies of name service data.

rpc_ns_mgmt_set_exp_age() Modifies the application's global expiration age for cached copies of name service data.

2.11.14 Local Management Services

The routines in this group provide a set of miscellaneous local operations that servers and clients can use to manage their RPC interactions.

rpc_mgmt_inq_com_timeout() Returns the communications time-out value referenced by a binding handle.

rpc_mgmt_set_authorization_fn() Establishes an authorisation function for processing remote calls to a server's management routines.

rpc_mgmt_set_cancel_timeout() Sets the lower bound on the time to wait before timing out after forwarding a cancel.

rpc_mgmt_set_com_timeout() Sets the communications time-out value referenced by a binding handle.

rpc_mgmt_set_server_stack_size() Specifies the stack size for each server thread.

rpc_mgmt_stats_vector_free() Frees a statistics vector.

rpc_mgmt_inq_dflt_protect_level() Returns the default protection level for an authentication service.

2.11.15 Local/Remote Management Services

Applications can use the routines in this group to query and stop servers remotely. Servers can also use these operations to query and stop themselves.

rpc_mgmt_inq_if_ids() Returns a vector of interface identifiers of the interfaces a server offers.

rpc_mgmt_inq_server_princ_name() Returns a server's principal name.

rpc_mgmt_inq_stats() Returns RPC run-time statistics.

rpc_mgmt_is_server_listening() Tells whether a server is listening for remote procedure calls.

rpc_mgmt_stop_server_listening() Tells a server to stop listening for remote procedure calls.

2.11.16 Error Messages

The **dce_error_inq_text**() routine provides a locale-independent way to get error message text for a status code returned by an RPC API routine. Because this routine is not RPC-specific, it is International Standard.d in Annex M rather than being included with the RPC API manual pages.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

© ISO/IEC

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 3. RPC API Manual Pages

3.1 RPC Data Types

The descriptions of the data types used by RPC API routines include C-language bindings, where appropriate.

The data type declarations given here fall into three categories:

- The declarations make use of a set of primitive unsigned integer data types. The C-language bindings for these types are implementation-dependent. Only the ranges of these types are given here.
- Certain data types are intended to be opaque to applications. The C-language bindings of opaque types are not given here.
- The remaining data types are defined explicitly here with C-language bindings that make use of the unsigned integer types, opaque types and other defined types.

Applications that refer to the data types described here must include the C header file **<dce/rpc.h>**.

3.1.1 Unsigned Integer Types

Some of RPC API function declarations and the remaining definitions given here make use of a set of unsigned integer data types. Each type holds an unsigned integer within a specified range, as shown in the following table.

Type Declaration	Range
unsigned8	0 to 2^8-1
unsigned16	0 to $2^{16}-1$
unsigned32	0 to $2^{32}-1$

The C-language bindings for these types are implementation-dependent.

3.1.2 Signed Integer Type

The **rpc_mgmt_set_cancel_timeout**() routine uses the **signed32** data type. This is an integer in the range -2^{31} to $2^{31}-1$.

3.1.3 Unsigned Character String

RPC treats all characters in strings as unsigned characters. The C-language binding of the unsigned character string type is implementation-dependent. The unsigned character data type must be able to encode the characters of the portable character set, as specified in Annex G. Routines that require character string arguments specify the data type **unsigned_char_t**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.4 Binding Handle

A binding handle is an opaque data type that applications use to reference binding information maintained by the RPC run-time system. Depending on the binding information that it references, a binding handle is considered a server binding handle or a client binding handle.

A server binding handle references binding information for a server. Server binding handles appear as arguments to many RPC API routines, and they are used both by clients and servers to manipulate the bindings required for remote procedure calls.

A client binding handle references binding information for a client that has made an RPC to a server. A client binding handle may be provided to the server application as the first argument to the call. (See Section 4 for further information.) Servers can use the routine **rpc_binding_server_from_client**() to convert a client binding handle to a server binding handle that can be used to make a remote procedure call to the calling client.

As described in Section 2, a binding handle refers to several components of binding information. When this binding information lacks an endpoint, the binding handle is said to be **partially bound**. When the binding information includes an endpoint, the binding handle is said to be **fully bound**. Both fully and partially bound binding handles can be used to make remote procedure calls.

RPC API routines requiring a binding handle as an argument specify the data type **rpc_binding_handle_t**. Binding handle arguments are passed by value.

The following table lists RPC API routines that operate on binding handles, specifying the type of binding handle required by each routine.

Table 3-1 Client and Server Binding Handles

Routine	Input Argument	Output Argument
rpc_binding_copy()	Server	Server
rpc_binding_free()	Server	None
rpc_binding_from_string_binding()	None	Server
rpc_binding_inq_auth_client()	Client	None
rpc_binding_inq_auth_info()	Server	None
rpc_binding_inq_object()	Server or client	None
rpc_binding_reset()	Server	None
rpc_binding_server_from_client()	Client	Server
rpc_binding_set_auth_info()	Server	None
rpc_binding_set_object()	Server	None
rpc_binding_to_string_binding()	Server or client	None
rpc_binding_vector_free()	Server	None
rpc_ns_binding_export()	Server	None
rpc_ns_binding_import_done()	None	Server
rpc_ns_binding_inq_entry_name()	Server	None
rpc_ns_binding_lookup_next()	None	Server
rpc_ns_binding_select()	Server	Server
rpc_server_inq_bindings()	None	Server

An application can share a single binding handle across multiple threads of execution. The application must provide concurrency control for operations that read or modify a shared binding handle. The related routines are:

rpc_binding_free()
rpc_binding_reset()
rpc_binding_set_auth_info()
rpc_binding_set_object()
rpc_ep_resolve_binding()
rpc_mgmt_set_com_timeout()

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.5 Binding Vector

The **binding vector** data structure contains a list of binding handles over which a server application can receive remote procedure calls.

The C-language declaration is:

```
typedef struct {
              unsigned32    count;
              rpc_binding_handle_t  binding_h[1];
              } rpc_binding_vector_t;
```

(The **[1]** subscript is a placeholder in the binding vector declaration. Applications use the **count** member to find the actual size of a returned binding vector.)

The RPC run-time system creates binding handles when a server application registers protocol sequences. To obtain a binding vector, a server application calls the **rpc_server_inq_bindings**( ) routine. A client application obtains a binding vector of compatible servers from the name service database by calling the **rpc_ns_binding_lookup_next**( ) routine. In both cases, the RPC run-time system allocates memory for the binding vector. An application calls the **rpc_binding_vector_free**( ) routine to free the binding vector.

To remove an individual binding handle from the vector, the application sets its value in the vector to NULL. When setting a vector element to NULL the application must:

- free the individual binding
- **not** change the value of **count**.

Calling the **rpc_binding_free**( ) routine allows an application both to free the unwanted binding handle and to set the vector entry to NULL.

The following routines require a binding vector argument:

rpc_binding_vector_free( )
rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_ep_unregister( )
rpc_ns_binding_export( )
rpc_ns_binding_lookup_next( )
rpc_ns_binding_select( )
rpc_server_inq_bindings( )

3.1.6 Boolean Type

Routines that require a Boolean-valued argument or return a Boolean value specify the data type **boolean32**. RPC implementations define the Boolean constants TRUE and FALSE.

3.1.7 Endpoint Map Inquiry Handle

An endpoint map inquiry handle is an opaque data type that references inquiry state information used by a series of endpoint inquiry operations. The endpoint inquiry handle data type is **rpc_ep_inq_handle_t**. Applications obtain an endpoint map inquiry handle by calling **rpc_mgmt_ep_elt_inq_begin**( ) and use the handle for one or more calls to **rpc_mgmt_ep_elt_inq_next**( ). Applications call **rpc_mgmt_ep_elt_inq_done**( ) to free an endpoint map handle.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.8 Interface Handle

Each stub declares an interface handle that can be used by application code to reference interface-related data maintained by the stub. The interface handle data type is **rpc_if_handle_t**. Applications refer to a stub-declared interface handle using a well-known name constructed as follows:

For the client:

*if-name*_v *major-version_minor-version*_c_ifspec

For the server:

*if-name*_v *major-version_minor-version*_s_ifspec

where:

- **if-name** is the interface identifier specified in the IDL file.
- **major-version** is the interface's major-version number specified in the IDL file.
- **minor-version** is the interface's minor-version number specified in the IDL file.

Implementations must support a maximum **if-name** length of at least 17 characters.

The following routines specify an interface handle argument:

rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ep_resolve_binding()
rpc_ep_unregister()
rpc_if_inq_id()
rpc_ns_binding_export()
rpc_ns_binding_import_begin()
rpc_ns_binding_lookup_begin()
rpc_ns_binding_unexport()
rpc_server_inq_if()
rpc_server_register_if()
rpc_server_unregister_if()
rpc_server_use_all_protseqs()
rpc_server_use_protseq_if()

3.1.9 Interface Identifier

An interface identifier (interface ID) data structure contains the interface UUID and major-version and minor-version numbers of an interface. The C-language declaration is:

```
typedef struct {
                uuid_t      uuid;
                unsigned16  vers_major;
                unsigned16  vers_minor;
                } rpc_if_id_t;
```

Applications can obtain an interface identifier by calling **rpc_if_inq_id**() with an interface handle. The following routines also require interface identifier arguments:

rpc_mgmt_ep_elt_inq_begin()
rpc_mgmt_ep_elt_inq_next()
rpc_mgmt_ep_unregister()
rpc_ns_mgmt_binding_unexport()
rpc_ns_profile_elt_add()
rpc_ns_profile_elt_inq_begin()
rpc_ns_profile_elt_inq_next()
rpc_ns_profile_elt_remove()

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.10 Interface Identifier Vector

The interface identifier (ID) vector data structure holds a list of interface identifiers.

The C-language declaration is:

```
typedef struct {
              unsigned32      count;
              rpc_if_id_t     *if_id[1];
              } rpc_if_id_vector_t;
```

(The **[1]** subscript is a placeholder in the interface ID vector declaration. Applications use the **count** member to find the actual size of a returned vector.)

To obtain a vector of the interface IDs registered by a server with the RPC run-time system, an application calls the **rpc_mgmt_inq_if_ids**( ) routine. To obtain a vector of the interface IDs exported by a server to a name service database, an application calls the **rpc_ns_mgmt_entry_inq_if_ids**( ) routine.

The RPC run-time system allocates memory for the interface ID vector. The application calls the **rpc_if_id_vector_free**( ) routine to free the interface ID vector.

3.1.11 Manager Entry Point Vector

The server stub declares a default manager entry point vector (EPV), which it uses to call the operations that implement an interface. A manager EPV consists of a vector of pointers to the operations of the interface. To declare the default manager EPV, the stub defines an interface-specific manager EPV data type with the following type name:

*<if-name>*_v*<major-version>*_*<minor-version>*_epv_t

The data type is defined as a C **struct** whose elements are pointers to the manager routines for the interface, with the same names and in the same order in which they appear in the IDL interface specification.

The stub declares the default manager EPV with the name **NIDL_manager_epv**.

Applications can use the stub-declared manager EPV data type to declare non-default manager EPVs. Applications initialise non-default manager EPVs with a vector of addresses of alternate manager routines. Applications that declare non-default manager EPVs must avoid the default name.

See **rpc_server_register_if**( ) for further information on non-default manager EPVs.

3.1.12 Name Service Handle

RPC API routines that obtain information from a name service use opaque name service handles to refer to search state information maintained by the run-time system. Applications obtain a name service handle by calling one of the name service **begin** routines and use the handle for one or more calls to the corresponding **next** routine. Applications free a name service handle by calling one of the name service **done** routines. For more information on name service handles and operations, refer to Section 2.

The name service handle data type is **rpc_ns_handle_t**.

The following routines require a name service handle argument:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_ns_binding_import_begin()
rpc_ns_binding_import_done()
rpc_ns_binding_import_done()
rpc_ns_binding_lookup_begin()
rpc_ns_binding_lookup_next()
rpc_ns_binding_lookup_done()
rpc_ns_entry_object_inq_begin()
rpc_ns_entry_object_inq_next()
rpc_ns_entry_object_inq_done()
rpc_ns_group_mbr_inq_begin()
rpc_ns_group_mbr_inq_next()
rpc_ns_group_mbr_inq_done()
rpc_ns_profile_elt_inq_begin()
rpc_ns_profile_elt_inq_next()
rpc_ns_profile_elt_inq_done()
rpc_ns_mgmt_handle_set_exp_age()

3.1.13 Protocol Sequence String

A protocol sequence string is a character string that identifies a protocol sequence. Protocol sequences are used to establish a relationship between a client and server. Valid protocol sequence strings are listed in Annex B. RPC applications should use only these strings.

Routines that require a protocol sequence string argument specify the data type **unsigned_char_t**.

Not all valid protocol sequences are supported by all implementations. An application can use a specific protocol sequence only if the implementation supports that protocol.

A server chooses to accept remote procedure calls over some or all of the supported protocol sequences. The following routines allow server applications to register protocol sequences with the run-time system:

rpc_server_use_all_protseqs()
rpc_server_use_all_protseqs()
rpc_server_use_protseq()
rpc_server_use_protseq_ep()
rpc_server_use_protseq_if()

Applications can use protocol sequence strings to construct string bindings using the **rpc_string_binding_compose**() routine.

3.1.14 Protocol Sequence Vector

The **protocol sequence vector** data structure contains a list of protocol sequence strings. The protocol sequence vector contains a **count** member followed by an array of pointers to protocol sequence strings.

The C-language declaration is:

```
typedef struct {
              unsigned32       count;
              unsigned_char_t  *protseq[1];
              } rpc_protseq_vector_t;
```

(The **[1]** subscript is a placeholder in the protocol sequence vector declaration. Applications use the **count** member to find the actual size of a returned binding vector.)

To obtain a protocol sequence vector, an application calls the **rpc_network_inq_protseqs**() routine. The RPC run-time system allocates memory for the protocol sequence vector. The application calls the **rpc_protseq_vector_free**() routine to free the protocol sequence vector.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.15 Statistics Vector

A **statistics vector** is used to store statistics from the RPC run-time system for a server instance. The statistics vector contains a **count** member followed by an array of statistics.

The C-language declaration is:

```
typedef struct {
                unsigned32          count;
                unsigned32          stats[1];
                } rpc_stats_vector_t, *rpc_stats_vector_p_t;
```

(The **[1]** subscript is a placeholder in the statistics vector declaration. Applications use the **count** member to find the actual size of a returned binding vector.)

This International Standard specifies four statistics that are returned in a statistics vector. The following constants are used to index the statistics array to extract specific statistics:

rpc_c_stats_calls_in The number of remote procedure calls received by the server.

rpc_c_stats_calls_out The number of remote procedure calls initiated by the server.

rpc_c_stats_pkts_in The number of RPC PDUs received by the server.

rpc_c_stats_pkts_out The number of RPC PDUs sent by the server.

To obtain run-time statistics, an application calls the **rpc_mgmt_inq_stats**( ) routine. The RPC run-time system allocates memory for the statistics vector. The application calls the **rpc_mgmt_stats_vector_free**( ) routine to free the statistics vector.

3.1.16 String Binding

A string binding contains the character representation of a binding handle.

The two formats of a string binding are shown below. The four italicised fields represent the object UUID, RPC protocol sequence, network address and endpoint and network options of the binding. A delimiter character such as an **@** (at sign) or a **:** (colon) separates each field. A string binding does not contain any white space.

object-uuid **@** *rpc-protocol-sequence* **:**
network-address **[***endpoint* **,** *option* **...**

or

object-uuid **@** *rpc-protocol-sequence* **:**
network-address **[endpoint =** *endpoint* **,** *option* **...**

object-uuid This field specifies an object UUID.

This field is optional. If it is not provided the RPC run-time system assumes a nil type UUID.

@ This symbol is the delimiter character for the object UUID field. If an object UUID is specified, it must be followed by this symbol.

rpc-protocol-sequence This field specifies a protocol sequence. Valid protocol sequence strings are listed in Annex B.

This field is required.

: This symbol is the delimiter character for the RPC protocol sequence field.

network-address This field specifies the address (**address**) of a host on a network (**network**) that receives remote procedure calls made with this string binding. The format and content of the network address depends on the value of **rpc-protocol-sequence**. For the internet protocols, the format for the network address is an optional # (number sign) character followed by four integers separated by periods.

The network address field is optional. If an application does not supply this field, the string binding refers to the local host.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

[	This symbol is the delimiter character specifying that one endpoint and zero or more options follow. If the string binding contains at least an endpoint, this symbol is required.
endpoint	This field specifies an endpoint of a specific server instance. Optionally the keyword **endpoint=** can precede the endpoint specifier. The format and content of the endpoint depends on the specified protocol sequence. For the internet protocols, the format of the endpoint field is a single integer. The endpoint field is optional. This symbol is the delimiter character specifying that option data follows. If an option follows, this delimiter is required.
option	This field specifies any options. Each option is specified as **option name=option value.** The format and content of the option depends on the specified protocol sequence. The option field is optional.
]	This symbol is the delimiter character specifying that one endpoint and zero or more options precede. If the string binding contains at least an endpoint, this symbol is required.

The \ (backslash) character is treated as an escape character for all string binding fields. It can be used to include one of the string delimiters in the value of a field.

3.1.17 String UUID

A **string UUID** contains the character representation of a UUID. A string UUID consists of multiple fields of hexadecimal characters. Dashes separate the fields and each field has a fixed length, as follows:

xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

For a detailed specification of string UUIDs, see Annex A.

The following routines require a string UUID argument:

rpc_string_binding_compose()
rpc_string_binding_parse()
uuid_from_string()
uuid_to_string()

3.1.18 UUIDs

Universal Unique Identifiers (UUIDs) are opaque data structures that are widely used by the RPC mechanism. The RPC API provides a series of routines to manipulate UUIDs. Routines that take a UUID argument declare the data type as **uuid_t**. (See Annex A for a detailed specification of UUIDs.)

3.1.19 UUID Vector

The UUID vector data structure contains a list of UUIDs. The UUID vector contains a **count** member, followed by an array of pointers to UUIDs.

The C-language declaration is:

```
typedef struct
{
    unsigned32      count;
    uuid_t          *uuid[1];
} uuid_vector_t;
```

The **[1]** subscript is a placeholder in the UUID vector declaration. Applications use the **count** member to find the actual size of a returned binding vector.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

An application constructs a UUID vector to contain object UUIDs to be exported or unexported from the name service database. The following routines require a UUID vector argument:

rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ep_unregister()
rpc_ns_binding_export()
rpc_ns_binding_unexport()
rpc_ns_mgmt_binding_unexport()

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_binding_copy — returns a copy of a binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_copy(
        rpc_binding_handle_t source_binding,
        rpc_binding_handle_t *destination_binding,
        unsigned32 *status);
```

ARGUMENTS

Input

source_binding Specifies the server binding handle whose referenced binding information will be copied.

Output

destination_binding Returns the server binding handle that refers to the copied binding information.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully or, if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_copy**( ) routine copies the server binding information referenced by the binding handle specified in the **source_binding** argument. This routine returns a new server binding handle for the copied binding information. The new server binding handle is returned in the **destination_binding** argument.

After calling this routine, operations performed on the **source_binding** binding handle do not affect the binding information referenced by the **destination_binding** binding handle. Similarly, operations performed on the **destination_binding** binding handle do not affect the binding information referenced by the **source_binding** binding handle.

Note: To release the memory used by the **destination_binding** binding handle and its referenced binding information, the application calls the **rpc_binding_free**( ) routine.

RETURN VALUE

None.

SEE ALSO

rpc_binding_free( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_free — releases binding handle resources

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_free(
       rpc_binding_handle_t *binding,
       unsigned32 *status);
```

ARGUMENTS

Input/Output

binding Specifies the server binding handle to free.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_free**() routine frees the memory used by a server binding handle and its referenced binding information when the binding handle was created by one of the following routines:

rpc_binding_copy()
rpc_binding_from_string_binding()
rpc_ns_binding_import_done()
rpc_ns_binding_select()
rpc_server_inq_bindings()
rpc_ns_binding_lookup_next()

When the operation succeeds, **binding** returns the value NULL.

RETURN VALUE

None.

SEE ALSO

rpc_binding_copy()
rpc_binding_from_string_binding()
rpc_ns_binding_import_done()
rpc_binding_vector_free()
rpc_ns_binding_lookup_next()
rpc_ns_binding_select()
rpc_server_inq_bindings().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_from_string_binding — returns a binding handle from a string representation of a binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_from_string_binding(
       unsigned_char_t *string_binding,
       rpc_binding_handle_t *binding,
       unsigned32 *status);
```

ARGUMENTS

Input

string_binding Specifies a string representation of a binding handle.

Output

binding Returns the server binding handle.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

DESCRIPTION

The **rpc_binding_from_string_binding**() routine creates a server binding handle from a string representation of a binding handle.

When the **string_binding** argument contains an object UUID, the returned binding contains the UUID that is specified. Otherwise, the returned **binding** contains a nil UUID.

When the **string_binding** argument contains an endpoint field, the returned **binding** is a fully bound server binding handle with a well-known endpoint. Otherwise, the returned **binding** is a partially bound binding handle.

When the **string_binding** argument contains a host address field, the returned **binding** contains the host address that is specified. Otherwise, the returned **binding** refers to the local host.

RETURN VALUE

None.

SEE ALSO

rpc_binding_copy()
rpc_binding_free()
rpc_binding_to_string_binding()
rpc_string_binding_compose().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_inq_auth_client — returns authentication, authorisation and protection information from a client binding handle

SYNOPSIS

```
#include <dce/rpc.h>
#include <dce/id_base.h>

void rpc_binding_inq_auth_client(
        rpc_binding_handle_t binding,
        rpc_authz_handle_t *privs,
        unsigned_char_t **server_princ_name,
        unsigned32 *protect_level,
        unsigned32 *authn_svc,
        unsigned32 *authz_svc,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies the client binding handle from which to return information.

Input/Output

server_princ_name Returns the server principal name referenced by **binding**. The content of the returned name and its syntax depend on the value of **authn_svc**. (See Annex D for authentication service-specific syntax.)

Specifying NULL prevents the routine from returning this argument.

Unless NULL is specified, the application should call the **rpc_string_free**() routine to free the storage used by this argument.

protect_level Returns the protection level referenced by **binding**. (See Annex D for possible values of this argument.)

Specifying NULL prevents the routine from returning this argument.

authn_svc Returns the authentication service referenced by **binding**. (See Annex D for possible values of this argument.)

Specifying NULL prevents the routine from returning this argument.

authz_svc Returns the authorisation service referenced by **binding**. (See Annex D for possible values of this argument.)

Specifying NULL prevents the routine from returning this argument.

Output

privs Returns a handle to the authorisation or privilege information referenced by **binding**.

The server must cast this handle to a data type that depends on **authz_svc**. (See Annex D for information about the data types appropriate to each authorisation service.)

The lifetime of the data referenced by this argument is one invocation of a server manager routine. If an application wants to preserve any of the returned data beyond this lifetime, it must copy the data into application-allocated memory.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

rpc_s_ok Success.

rpc_s_binding_has_no_auth
Binding has no authentication information.

DESCRIPTION

The **rpc_binding_inq_auth_client**( ) routine returns authentication, authorisation and privilege information referenced by the client binding handle, **binding**. Servers obtain client binding handles as the first argument of a remote procedure call. (See Subclause 3.1 and Section 2 for more detailed information on how client binding handles are created and obtained.) The client binding handle references authentication, authorisation and privilege information for the client that made the remote procedure call.

A client establishes this information by calling **rpc_binding_set_auth_info**( ), which associates a set of authentication, authorisation and privilege information with a server binding handle. When the client makes an RPC call on this server binding handle, the client binding handle received by the server references the same authentication, authorisation and privilege information.

No server memory is allocated for the data referenced by **privs**. The lifetime of this data is the current invocation of the manager routine that was called with the **binding** argument. An application that wishes to preserve any privileges information beyond this invocation must copy the information into server memory.

RETURN VALUE

None.

SEE ALSO

rpc_binding_inq_auth_info( )
rpc_binding_set_auth_info( )
rpc_string_free( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_inq_auth_info — returns authentication, authorisation and protection information from a server binding handle

SYNOPSIS

```
#include <dce/rpc.h>
#include <dce/sec_login.h>

void rpc_binding_inq_auth_info(
        rpc_binding_handle_t binding,
        unsigned_char_t **server_princ_name,
        unsigned32 *protect_level,
        unsigned32 *authn_svc,
        rpc_auth_identity_handle_t *auth_identity,
        unsigned32 *authz_svc,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies the server binding handle from which to return information.

Input/Output

server_princ_name Returns the server principal name referenced by **binding**. The content of the returned name and its syntax depend on the value of **authn_svc**. (See Annex D for authentication service-specific syntax.)

Specifying NULL prevents the routine from returning this argument.

Unless NULL is specified, the application should call the **rpc_string_free**() routine to free the storage used by this argument.

protect_level Returns the protection level referenced by **binding**. (See Annex D for possible values of this argument.)

Specifying NULL prevents the routine from returning this argument.

authn_svc Returns the authentication service referenced by **binding**. (See Annex D for possible values of this argument.)

Specifying NULL prevents the routine from returning this argument.

auth_identity Returns a handle to a data structure that contains the client's authentication and authorisation credentials. This argument must be cast as appropriate for the authentication and authorisation services specified by **authn_svc** and **authz_svc**. (See Annex D for information about the appropriate data types appropriate to each service.)

Specifying NULL prevents the routine from returning this argument.

authz_svc Returns the authorisation service referenced by **binding**. (See Annex D for possible values of this argument.)

Specifying NULL prevents the routine from returning this argument.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_s_binding_has_no_auth
Binding has no authentication information.

DESCRIPTION

The **rpc_binding_inq_auth_info**( ) routine returns authentication, authorisation and protection-level information referenced by the server binding handle, **binding**. Client applications use this routine to discover whether the protection level they have requested is supported by the RPC run-time implementation.

A client application associates authentication, authorisation and protection-level information with a server binding handle by calling **rpc_binding_set_auth_info**( ). The value of **protect_level** returned by **rpc_binding_inq_auth_info**( ) may be higher than the level requested in the previous call to **rpc_binding_set_auth_info**( ). When an application requests a protection level that is not supported, the RPC run-time system attempts to upgrade the protection level to the next highest supported level. When it succeeds, the binding will be given a higher protection level than the one requested. Client applications may compare the requested protection level with the value returned by **rpc_binding_inq_auth_info**( ) to discover whether the requested protection level is actually supported by the run-time system.

The **auth_identity** argument points to the authentication and authorisation identity information associated with **binding**. **rpc_binding_inq_auth_info**( ) allocates no memory for this information, and references to **auth_identity** may not be valid after any subsequent call to **rpc_binding_set_auth_info**( ) with the same **binding** argument. In any case, the lifetime of **auth_identity** is no longer than the lifetime of **binding**.

Any of the data returned by **rpc_binding_inq_auth_info**( ) may be stale after a subsequent call to **rpc_binding_set_auth_info**( ) with the same **binding** argument.

The **rpc_binding_inq_auth_info**( ) routine allocates memory for the returned **server_princ_name** argument. The caller is responsible for calling the **rpc_string_free**( ) routine for the returned argument string.

RETURN VALUE

None.

SEE ALSO

rpc_binding_set_auth_info( )
rpc_string_free( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_inq_object — returns the object UUID from a binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_inq_object(
        rpc_binding_handle_t binding,
        uuid_t *object_uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a client or server binding handle.

Output

object_uuid Returns the object UUID found in the **binding** argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_inq_object**() routine obtains the object UUID associated with a binding handle. If no object UUID is associated with the binding handle, this routine returns a nil UUID.

RETURN VALUE

None.

SEE ALSO

rpc_binding_set_object().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_reset — resets a binding handle to a partially bound binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_reset(
        rpc_binding_handle_t binding,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies the server binding handle to reset.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_reset**() routine removes the endpoint portion of the server address referenced by the binding handle, **binding**. The result is a partially bound server binding handle.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register()
rpc_ep_register_no_replace().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_server_from_client — converts a client binding handle to a server binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_server_from_client(
        rpc_binding_handle_t client_binding,
        rpc_binding_handle_t *server_binding,
        unsigned32 *status);
```

ARGUMENTS

Input

client_binding Specifies the client binding handle to convert to a server binding handle.

Output

server_binding Returns a server binding handle.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_server_from_client**() routine converts a client binding handle into a partially bound server binding handle.

An application obtains a client binding handle as an argument passed to a server manager routine from the RPC run-time system during a remote procedure call. When an RPC arrives at a server, the RPC run-time system creates a client binding handle that contains binding information about the calling client host. The run-time system passes the client binding handle to the server manager routine as the first argument. The argument type is **rpc_binding_handle_t**.

The server binding handle returned from **rpc_binding_server_from_client**() references binding information that is constructed as follows:

- It contains a network address for the calling client's host but lacks an endpoint. The returned binding handle is thus partially bound.
- It contains the same object UUID used by the calling client. This may be the nil UUID. (See **rpc_binding_set_object**(), **rpc_ns_binding_import_begin**(), **rpc_ns_binding_lookup_begin**() and **rpc_binding_from_string_binding**() to see how a client specifies an object UUID for a call.)
- It contains no authentication information.

RETURN VALUE

None.

SEE ALSO

rpc_binding_free()
rpc_binding_set_object
rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ns_binding_import_begin()
rpc_ns_binding_lookup_begin()
rpc_binding_from_string_binding().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_set_auth_info — sets authentication, authorisation and protection-level information for a binding handle

SYNOPSIS

```
#include <dce/rpc.h>
#include <dce/sec_login.h>

void rpc_binding_set_auth_info(
        rpc_binding_handle_t binding,
        unsigned_char_t *server_princ_name,
        unsigned32 protect_level,
        unsigned32 authn_svc,
        rpc_auth_identity_handle_t auth_identity,
        unsigned32 authz_svc,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies the server binding handle for which to set the authentication, authorisation and protection-level information.

server_princ_name Specifies a principal name for the server referenced by **binding**. The content and syntax of this name depend on the value of **authn_svc**. (See Annex D for authentication service-specific syntax.)

Note: An application can call the **rpc_mgmt_inq_server_princ_name**( ) routine to obtain the principal name of a server that is registered for the required authentication service. (See **rpc_mgmt_inq_server_princ_name**( ) for details.)

protect_level Specifies the protection level for remote procedure calls made using **binding**. The protection level determines the degree to which authenticated communications between the client and the server are protected. (See Annex D for possible values of this argument.)

authn_svc Specifies the authentication service to use for calls made on **binding**. (See Annex D for possible values of this argument.)

auth_identity Specifies a handle for a data structure that contains the client's authentication and authorisation credentials. The data type of this structure depends on the values of **authn_svc** and **authz_svc**. (See Annex D for information on the service-specific data types.)

Specify NULL to use the default security login context for the current address space. The default is the context in effect at the time of the call to **rpc_binding_set_auth_info**( ). For information on how the default security login context is established, you can refer to the **DCE: Security Services** specification when it becomes available.

authz_svc Specifies the authorisation service to be used for calls made on **binding**. (See Annex D for possible values of this argument.)

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unknown_authn_service
Unknown authentication service.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_s_authn_authz_mismatch
: The requested authorisation service is not supported by the requested authentication service.

rpc_s_unsupported_protect_level
: The requested protection level is not supported and could not be upgraded to a higher supported level.

rpc_s_proto_unsupp_by_auth
: RPC protocol is not supported by the requested authentication protocol

rpc_s_no_princ_name
: No principal name is registered.

rpc_s_not_authorized
: Not authorised for operation.

DESCRIPTION

The **rpc_binding_set_auth_info**( ) routine sets authentication, authorisation and protection-level information for the server binding handle, **binding**. A client application that wants to make authenticated remote procedure calls first calls this routine. Any RPC calls subsequently made on **binding** will be authenticated according to the information set by this call. If a client application has not called **rpc_binding_set_auth_info**( ) for a binding, remote procedure calls made on that binding are unauthenticated.

Note that the value of **protect_level** actually set for **binding** depends on the protection levels supported by the implementation. The value set may be higher than the level requested. When an application requests a protection level that is not supported, the RPC run-time system attempts to upgrade the protection level to the next highest supported level. When it succeeds, the binding will be given a higher protection level than the one requested. Annex D gives the canonical ordering of **protect_level** values from lowest to highest. Applications can call the routine **rpc_binding_inq_auth_info**( ) to discover the protection level actually set.

To find the authentication, authorisation and protection-level information set for a binding handle, applications call **rpc_binding_inq_auth_info**( ).

RETURN VALUE

None.

SEE ALSO

rpc_binding_inq_auth_info( )
rpc_mgmt_inq_server_princ_name( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_set_object — sets the object UUID value in a binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_set_object(
        rpc_binding_handle_t binding,
        uuid_t *object_uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies the server binding into which argument **object_uuid** is set.

object_uuid Specifies the UUID of the object serviced by the server specified in the **binding** argument.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_set_object**() routine associates an object UUID with a server binding handle. This operation replaces the previously associated object UUID with the UUID in the **object_uuid** argument.

RETURN VALUE

None.

SEE ALSO

rpc_binding_from_string_binding()
rpc_binding_inq_object().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_to_string_binding — returns a string representation of a binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_to_string_binding(
        rpc_binding_handle_t binding,
        unsigned_char_t **string_binding,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a client or server binding handle to convert to a string representation of a binding handle.

Output

string_binding Returns a pointer to the string representation of the binding handle specified in the **binding** argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_to_string_binding**() routine converts a client or server binding handle to its string representation.

The RPC run-time system allocates memory for the string returned in the **string_binding** argument. The application calls the **rpc_string_free**() routine to deallocate that memory.

When the binding handle in the **binding** argument contains a nil object UUID, the object UUID field is not included in the returned string.

RETURN VALUE

None.

SEE ALSO

rpc_binding_from_string_binding()
rpc_string_binding_parse()
rpc_string_free().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_binding_vector_free — frees the memory used to store a vector of binding handles

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_binding_vector_free(
        rpc_binding_vector_t **binding_vector,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

binding_vector Specifies the address of a pointer to a vector of server binding handles.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_binding_vector_free**() routine frees the memory used to store a vector of server binding handles when the vector was created using either the **rpc_server_inq_bindings**() routine or **rpc_ns_binding_lookup_next**() routine. The freed memory includes both the binding handles and the vector itself.

The **rpc_binding_free**() routine may be used to free individual elements of the vector. When an element has been freed with this routine, the NULL element entry replaces it; the **rpc_binding_vector_free**() routine ignores such an entry.

When the **rpc_binding_vector_free**() routine succeeds, the **binding_vector** pointer is set to NULL.

RETURN VALUE

None.

SEE ALSO

rpc_binding_free()
rpc_server_inq_bindings()
rpc_ns_binding_lookup_next().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ep_register — adds to, or replaces, server address information in the local endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ep_register(
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned_char_t *annotation,
        unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies an interface specification to register with the local endpoint map.

binding_vec Specifies a vector of server binding handles over which the server can receive remote procedure calls.

object_uuid_vec Specifies a vector of object UUIDs that the server offers. The server application constructs this vector.

The application supplies the value NULL to indicate that there are no object UUIDs to register. In this case, each cross-product element added to the local endpoint map contains the nil UUID. (See **DESCRIPTION** for further discussion of cross-product elements.)

annotation Defines a character string comment applied to each cross-product element added to the local endpoint map. The string can be up to 64 characters long, including the null terminating character. Strings longer than 64 characters are truncated. The application supplies the value NULL or the string "" to indicate an empty annotation string.

When replacing elements, the annotation string supplied, including an empty annotation string, replaces any existing annotation string.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

ept_s_cant_perform_op
Cannot perform the requested operation.

DESCRIPTION

The **rpc_ep_register**( ) routine adds elements to, or replaces elements in, the local host's endpoint map.

Each element added to the local endpoint map logically contains the following:

- interface ID, consisting of an interface UUID and versions (major and minor)
- binding information
- object UUID, which may be the nil UUID
- annotation, which may be an empty string.

When an existing map element matches a supplied element, this routine replaces the map element's endpoint with the endpoint from the supplied element's binding information. When there is no such match, a new map element is added.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

For a match between an existing and supplied element to occur, the interface UUIDs, object UUIDs and binding information (except for the endpoint) from both elements must be equal. Matching rules for interface version numbers are specified in the following table.

Existing Element	Relationship	Supplied Element	Routine's Action
Major version number	Not equal to	Major version number	Ignores minor version number relationship and adds a new endpoint map element. The existing element remains unchanged.
Major version number	Equal to	Major version number	Acts according to the minor version number relationship.
Minor version number	Equal to	Minor version number	Replaces the endpoint of the existing element based on the supplied information.
Minor version number	Less than	Minor version number	Replaces the existing element based on the supplied information.
Minor version number	Greater than	Minor version number	Ignores the supplied information. The existing element remains unchanged.

A server uses this routine when only a single instance of the server will run on the server's host; that is, when no more than one server instance will offer the same interface UUID, object UUID and protocol sequence. Servers use **rpc_ep_register_no_replace**( ) when multiple instances of the server may run on the server's host.

Note: Servers should call **rpc_ep_unregister**( ) to unregister endpoints before they stop running. If a server stops running without calling **rpc_ep_unregister**( ), applications may waste time trying to communicate with the nonexistent server. Since **rpc_ep_register**( ) replaces existing compatible local endpoint map elements, it will remove obsolete compatible elements left by servers that have crashed without unregistering their endpoints. However, server applications that stop normally should unregister their endpoints. They should not rely on new instantiations to clean up obsolete endpoints

A server application calls this routine to register endpoints that have been specified by calling any of the following routines:

rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )

Note: When the server also exports binding information to the name service database, the server calls this routine with the same **if_handle**, **binding_vec**, and **object_uuid_vec** arguments that the server uses when calling the **rpc_ns_binding_export**( ) routine.

The **rpc_ep_register**( ) routine creates elements to add to the local endpoint map as a cross-product of the **if_handle**, **binding_vec** and **object_uuid_vec** arguments.

When the **object_uuid_vec** argument is NULL, the cross-product of **if_handle**, **binding_vec** and the nil UUID is created.

The annotation string is also included in each cross-product element. The string is used by applications for informational purposes only. The RPC run-time system does not use it to determine which server instance a client communicates with, or for enumerating endpoint map elements.

The following example shows the cross-product created when **if_handle** has the value ifhand, **binding_vec** has the values b1, b2, b3, and **object_uuid_vec** has the values u1, u2, u3, u4. The cross-product contains 12 elements, as follows:

```
(ifhand,b1,u1)  (ifhand,b1,u2)  (ifhand,b1,u3)  (ifhand,b1,u4)
(ifhand,b2,u1)  (ifhand,b2,u2)  (ifhand,b2,u3)  (ifhand,b2,u4)
(ifhand,b3,u1)  (ifhand,b3,u2)  (ifhand,b3,u3)  (ifhand,b3,u4)
```

Each cross-product element also contains the annotation string.

RETURN VALUE

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SEE ALSO

rpc_ep_register_no_replace()
rpc_ep_resolve_binding()
rpc_ep_unregister()
rpc_mgmt_ep_unregister()
rpc_ns_binding_export()
rpc_server_inq_bindings()
rpc_server_use_all_protseqs()
rpc_server_use_all_protseqs()
rpc_server_use_protseq()
rpc_server_use_protseq_ep()
rpc_server_use_protseq_if().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ep_register_no_replace — adds to server address information in the local endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ep_register_no_replace(
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned_char_t *annotation,
        unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies an interface specification to register with the local endpoint map.

binding_vec Specifies a vector of binding handles over which the server can receive remote procedure calls.

object_uuid_vec Specifies a vector of object UUIDs that the server offers.

The application supplies the value NULL to indicate there are no object UUIDs to register. In this case, each cross-product element contains the nil UUID.

annotation Defines a character string comment applied to each cross-product element added to the local endpoint map. The string can be up to 64 characters long, including the null-terminating character. If the application specifies an empty string (""), each cross-product element will contain an empty string.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

ept_s_cant_perform_op
Cannot perform requested operation.

DESCRIPTION

The **rpc_ep_register_no_replace**( ) routine adds elements to the local host's endpoint map. The routine does not replace existing elements. Otherwise, this routine is identical to routine **rpc_ep_register**( ). A server application uses this routine, instead of routine **rpc_ep_register**( ), when multiple instances of the server run on the same host. Servers should use this routine if, at any time, more than one server instance offers the same interface UUID, object UUID, and protocol sequence.

Note: Servers should call **rpc_ep_unregister**( ) before they stop running to remove their endpoints from the local endpoint map. When obsolete elements are left in the endpoint map, clients may waste time trying to communicate with non-existent servers. Obsolete elements, left by servers that have stopped without calling **rpc_ep_unregister**( ), are periodically removed from the local endpoint map. However, during the time between these removals, the obsolete elements increase the chance that a client will attempt to communicate with a non-existent server.

A server program calls this routine to register endpoints that were specified by calling any of the following routines:

rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Note: If the server also exports to the name service database, the server calls this routine with the same **if_handle**, **binding_vec** and **object_uuid_vec** arguments as the server uses when calling the **rpc_ns_binding_export**( ) routine.

The **rpc_ep_register** routine creates elements to add to the local endpoint map as a cross-product of the **if_handle**, **binding_vec** and **object_uuid_vec** arguments.

When the **object_uuid_vec** argument is NULL, the cross-product of **if_handle**, **binding_vec** and the nil type UUID is created.

The annotation string is also included in each cross-product element. The string is used by applications for informational purposes only. The RPC run-time system does not use it to determine which server instance a client communicates with, or for enumerating endpoint map elements.

rpc_ep_register( ) contains an example of a cross-product.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register( )
rpc_ep_resolve_binding( )
rpc_ep_unregister( )
rpc_mgmt_ep_unregister( )
rpc_ns_binding_export( )
rpc_server_inq_bindings( )
rpc_server_use_all_protseqs( )
rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )
rpc_server_use_protseq_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ep_resolve_binding — resolves a partially bound server binding handle into a fully bound server binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ep_resolve_binding(
        rpc_binding_handle_t binding,
        rpc_if_handle_t if_handle,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

binding Specifies a partially bound server binding handle to resolve into a fully bound server binding handle.

if_handle Contains a stub-generated data structure that specifies the interface of interest.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

ept_s_not_registered
No entries found.

DESCRIPTION

An application calls the **rpc_ep_resolve_binding**( ) routine to resolve a partially bound server binding handle into a fully bound server binding handle.

To resolve a binding, **rpc_ep_resolve_binding**( ) obtains an endpoint for a compatible server instance from the endpoint map of the host specified by **binding**. In selecting an endpoint, **rpc_ep_resolve_binding**( ) uses the interface UUID associated with **if_handle** and the object UUID associated with **binding**. The object UUID may be the nil UUID. The endpoint matching algorithm is described in **rpc_ep_register**( ).

The resolved binding returned by **rpc_ep_resolve_binding**( ) depends on whether the specified binding handle is partially bound or fully bound. When the application specifies a partially bound handle, the routine produces the following results:

- If no compatible server instances are registered in the endpoint map, the routine returns the ept_s_not_registered status code.
- If one compatible server instance is registered in the local endpoint map, the routine returns a fully bound binding handle in **binding** and the rpc_s_ok status code.
- If more than one compatible server instance is registered in the local endpoint map, the routine arbitrarily selects one. It then returns the corresponding fully bound binding handle in **binding** and the rpc_s_ok status code.

When the application specifies a fully bound binding handle, the routine returns the specified binding handle in **binding** and the rpc_s_ok status code.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register( )
rpc_ep_register_no_replace( )

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_mgmt_ep_elt_inq_begin()
rpc_mgmt_ep_elt_inq_done()
rpc_mgmt_ep_elt_inq_next()
rpc_binding_from_string_binding()
rpc_binding_reset().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ep_unregister — removes server address information from the endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ep_unregister(
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies an interface specification to remove (that is, unregister) from the endpoint map.

binding_vec Specifies a vector of binding handles to remove.

object_uuid_vec Specifies a vector of object UUIDs to remove. The server application constructs this vector. When the value NULL is supplied, the routine constructs the cross-product of **if_handle** and **binding_vec** with the nil object UUID.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

ept_s_cant_perform_op
Cannot perform requested operation.

DESCRIPTION

An application calls **rpc_ep_unregister**( ) to remove endpoint map elements that it has previously registered.

Note: The application calls the **rpc_server_inq_bindings**( ) routine to obtain the required **binding_vec** argument. To remove selected endpoints, the application can remove individual elements from argument **binding_vec** before calling this routine.

This routine creates a cross-product from the **if_handle**, **binding_vec** and **object_uuid_vec** arguments, and removes each element that matches the cross-product from the local endpoint map. **rpc_ep_register**( ) discusses the construction of the cross-product.

Matches to elements in the endpoint map are exact. In particular, cross-product elements containing the nil object UUID only match elements in the endpoint map that contain the nil object UUID. Therefore, specifying NULL for the **uuid_vec** argument results in removing only elements with the nil object UUID from the endpoint map.

Note: Servers should call **rpc_ep_unregister**( ) to unregister their endpoints before they stop running. If they fail to do so, clients may find the obsolete endpoints and waste time trying to communicate with the non-existent servers.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_mgmt_ep_unregister( )
rpc_ns_binding_unexport( )
rpc_server_inq_bindings( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_if_id_vector_free — frees a vector and the interface identifier structures it contains

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_if_id_vector_free(
      rpc_if_id_vector_t **if_id_vector,
      unsigned32 *status);
```

ARGUMENTS

Input/Output

if_id_vector Specifies the address of a pointer to a vector of interface information. On success this argument is set to NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_if_id_vector_free**() routine frees the memory used to store a vector of interface identifiers when they have been obtained by calling either **rpc_ns_mgmt_entry_inq_if_ids**() or **rpc_mgmt_inq_if_ids**(). This freed memory includes memory used by the interface identifiers and the vector itself.

RETURN VALUE

None.

SEE ALSO

rpc_if_inq_id()
rpc_mgmt_inq_if_ids()
rpc_ns_mgmt_entry_inq_if_ids().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_if_inq_id — returns the interface identifier for an interface specification

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_if_inq_id(
       rpc_if_handle_t if_handle,
       rpc_if_id_t *if_id,
       unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies the interface specification to inquire about.

Output

if_id Pointer to the returned interface identifier. The application provides memory for the returned data.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

Applications call the **rpc_if_inq_id**( ) routine to obtain the interface identifier from the provided interface specification. Subclause 3.1 specifies how applications can construct the name of a stub-declared interface handle.

RETURN VALUE

None.

SEE ALSO

rpc_if_id_vector_free( )
rpc_mgmt_inq_if_ids( )
rpc_ns_mgmt_entry_inq_if_ids( )

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_ep_elt_inq_begin — creates an inquiry context for viewing the elements in an endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_ep_elt_inq_begin(
        rpc_binding_handle_t ep_binding,
        unsigned32 inquiry_type,
        rpc_if_id_t *if_id,
        unsigned32 vers_option,
        uuid_t *object_uuid,
        rpc_ep_inq_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input

ep_binding Specifies the host whose endpoint map elements will be viewed.

To view elements from the local host, the application specifies NULL.

To view endpoint map elements from another host, the application specifies a server binding handle for that host. The object UUID associated with this argument must be a nil UUID. When a non-nil UUID is specified, the routine fails with the status code ept_s_cant_perform_op.

inquiry_type An integer value that indicates the type of inquiry to perform on the endpoint map. The following list presents the valid inquiry types:

Value	Description
rpc_c_ep_all_elts	Returns every element from the endpoint map. The **if_id**, **vers_option** and **object_uuid** arguments are ignored.
rpc_c_ep_match_by_if	Searches the endpoint map for those elements that contain the interface identifier specified by the **if_id** and **vers_option** values. The **object_uuid** argument is ignored.
rpc_c_ep_match_by_obj	Searches the endpoint map for those elements that contain the object UUID specified by the **object_uuid** argument. The **if_id** and **vers_option** arguments are ignored.
rpc_c_ep_match_by_both	Searches the endpoint map for those elements that contain the interface identifier and object UUID specified by the **if_id**, **vers_option** and **object_uuid** arguments.

if_id Specifies the interface identifier of the endpoint map elements to be returned by the **rpc_mgmt_ep_elt_inq_next** routine.

This argument is meaningful only when **inquiry_type** is one of rpc_c_ep_match_by_if or rpc_c_ep_match_by_both. Otherwise, the argument is ignored.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

vers_option Specifies how the **rpc_mgmt_ep_elt_inq_next**() routine uses the **if_id** argument.

This argument is meaningful only when **inquiry_type** is one of rpc_c_ep_match_by_if or rpc_c_ep_match_by_both. Otherwise, this argument is ignored.

The following list presents the valid values for this argument.

Value	Description
rpc_c_vers_all	Returns endpoint map elements that offer the specified interface UUID, regardless of the version numbers.
rpc_c_vers_compatible	Returns endpoint map elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.
rpc_c_vers_exact	Returns endpoint map elements that offer the specified version of the specified interface UUID.
rpc_c_vers_major_only	Returns endpoint map elements that offer the same major version of the specified interface UUID (ignores the minor version).
rpc_c_vers_upto	Returns endpoint map elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version.

object_uuid Specifies the object UUID that the **rpc_mgmt_ep_elt_inq_next**() routine looks for in endpoint map elements.

This argument is meaningful only when **inquiry_type** is one of rpc_c_ep_match_by_obj or rpc_c_ep_match_by_both. Otherwise, this argument is ignored.

Output

inquiry_context Returns an inquiry context for use with the **rpc_mgmt_ep_elt_inq_next**() and **rpc_mgmt_ep_elt_inq_done**() routines.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok	Success.

DESCRIPTION

The **rpc_mgmt_ep_elt_inq_begin**() routine creates an inquiry context for viewing server address information stored in the endpoint map.

Using the **inquiry_type** and **vers_option** arguments, an application specifies which of the following endpoint map elements are to be returned from calls to the **rpc_mgmt_ep_elt_inq_next**() routine:

- all elements
- those elements with the specified interface identifier
- those elements with the specified object UUID

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- those elements with both the specified interface identifier and object UUID.

Before calling the **rpc_mgmt_ep_elt_inq_next**( ) routine, the application must first call this routine to create an inquiry context.

After viewing the endpoint map elements, the application calls the **rpc_mgmt_ep_elt_inq_done**( ) routine to delete the inquiry context.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_ep_unregister( )
rpc_mgmt_ep_elt_inq_done( )
rpc_mgmt_ep_elt_inq_next( )
rpc_mgmt_ep_unregister( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_ep_elt_inq_done — deletes the inquiry context for viewing the elements in an endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_ep_elt_inq_done(
        rpc_ep_inq_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

inquiry_context Specifies the inquiry context to delete.

Returns the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_mgmt_ep_elt_inq_done**( ) routine deletes an inquiry context created by the **rpc_mgmt_ep_elt_inq_begin**( ) routine.

An application calls this routine after viewing local endpoint map elements using the **rpc_mgmt_ep_elt_inq_next**( ) routine.

RETURN VALUE

None.

SEE ALSO

rpc_mgmt_ep_elt_inq_begin( )
rpc_mgmt_ep_elt_inq_next( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_ep_elt_inq_next — returns one element from an endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_ep_elt_inq_next(
        rpc_ep_inq_handle_t inquiry_context,
        rpc_if_id_t *if_id,
        rpc_binding_handle_t *binding,
        uuid_t *object_uuid,
        unsigned_char_t **annotation,
        unsigned32 *status);
```

ARGUMENTS

Input

inquiry_context Specifies an inquiry context. This inquiry context is returned from the **rpc_mgmt_ep_elt_inq_begin** routine.

Output

if_id Returns the interface identifier of the endpoint map element.

binding Returns the binding handle from the endpoint map element.

Specify NULL to prevent the routine from returning this argument.

object_uuid Returns the object UUID from the endpoint map element.

Specify NULL to prevent the routine from returning this argument.

annotation Returns the annotation string for the endpoint map element. When there is no annotation string in the endpoint map element, the empty string ("") is returned.

Specify NULL to prevent the routine from returning this argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

ept_s_cant_perform_op
Cannot perform the requested operation.

rpc_s_no_more_elements
No more elements.

rpc_s_com_failure
Communications failure.

DESCRIPTION

The **rpc_mgmt_ep_elt_inq_next**() routine returns one element from the endpoint map. Elements selected depend on the inquiry context. The selection criteria are determined by the **inquiry_type** argument of the **rpc_mgmt_ep_elt_inq_begin**() call that returned **inquiry_context**. **rpc_mgmt_ep_elt_inq_begin**() describes inquiry types.

An application can view all the selected endpoint map elements by repeatedly calling the **rpc_mgmt_ep_elt_inq_next**() routine. When all the elements have been viewed, this routine returns an rpc_s_no_more_elements status. The returned elements are unordered.

When the respective arguments are non-NULL, the RPC run-time system allocates memory for the returned **binding** and the **annotation** string on each call to this routine. The application is responsible for calling the **rpc_binding_free**() routine for each returned **binding** and the **rpc_string_free**()

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

routine for each returned **annotation** string.

After viewing the endpoint map's elements, the application must call the **rpc_mgmt_ep_elt_inq_done**( ) routine to delete the inquiry context.

RETURN VALUE

None.

SEE ALSO

rpc_binding_free( )
rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_mgmt_ep_elt_inq_begin( )
rpc_mgmt_ep_elt_inq_done( )
rpc_string_free( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_ep_unregister — removes server address information from an endpoint map

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_ep_unregister(
        rpc_binding_handle_t ep_binding,
        rpc_if_id_t *if_id,
        rpc_binding_handle_t binding,
        uuid_t *object_uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

ep_binding Specifies the host whose endpoint map elements are to be unregistered. To remove elements from the same host as the calling application, the application specifies NULL.

To remove endpoint map elements from another host, the application specifies a server binding handle for any server residing on that host.

Note: The application can specify the same binding handle it is using to make other remote procedure calls.

if_id Specifies the interface identifier to remove from the endpoint map.

binding Specifies the binding handle to remove.

object_uuid Specifies an optional object UUID to remove.

The value NULL indicates there is no object UUID to remove.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

ept_s_cant_perform_op
Cannot perform the requested operation.

rpc_s_comm_failure
Communications failure.

ept_s_not_registered
No entries found.

DESCRIPTION

The **rpc_mgmt_ep_unregister**( ) routine unregisters an element from an endpoint map. A management program calls this routine to remove addresses of servers that are no longer available, or to remove addresses of servers that support objects that are no longer offered.

The **ep_binding** argument must be a full binding. The object UUID associated with the **ep_binding** argument must be a nil UUID. Specifying a non-nil UUID causes the routine to fail with the status code ept_s_cant_perform_op. Other than the host information and object UUID, all information in this argument is ignored.

Note: Use this routine cautiously. Removing elements from the local endpoint map may make servers unavailable to client applications that do not already have a fully bound binding handle to the server.

An application calls the **rpc_mgmt_ep_elt_inq_next**( ) routine to view local endpoint map elements. The application can then remove the elements using the **rpc_mgmt_ep_unregister**( )

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

routine.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ns_binding_unexport()
rpc_mgmt_ep_elt_inq_begin()
rpc_mgmt_ep_elt_inq_done()
rpc_mgmt_ep_elt_inq_next().

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_inq_com_timeout — returns the communications timeout value for a server binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_inq_com_timeout(
        rpc_binding_handle_t binding,
        unsigned32 *timeout,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a server binding handle.

Output

timeout Returns the communications timeout value from the **binding** argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_mgmt_inq_com_timeout**( ) routine returns the communications timeout value in a server binding handle.

rpc_mgmt_set_com_timeout( ) explains the timeout values in the returned **timeout**.

To change the timeout value, a client calls the **rpc_mgmt_set_com_timeout**( ) routine.

RETURN VALUE

None.

SEE ALSO

rpc_mgmt_set_com_timeout( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_inq_dflt_protect_level — returns the default protection level for an authentication service

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_inq_dflt_protect_level(
       unsigned32 authn_svc,
       unsigned32 *protect_level,
       unsigned32 *status);
```

ARGUMENTS

Input

authn_svc Specifies the authentication service for which to return the default protection level. (See Annex D for values of this argument.)

Output

protect_level Returns the default protection level for the specified authentication service. The protection level determines the degree to which authenticated communications between the client and the server are protected. (See Annex D for values of this argument.)

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unknown_auth_service
Unknown authentication service.

DESCRIPTION

The **rpc_mgmt_inq_dflt_protect_level**() routine returns the default protection level for the specified authentication service. The **protect_level** value returned is the same as the value implied when the application calls the **rpc_binding_set_auth_info**() or **rpc_server_register_auth_info**() routines with the same **authn_svc** value and the **protect_level** value of rpc_c_protect_level_default.

RETURN VALUE

None.

SEE ALSO

rpc_binding_inq_auth_client()
rpc_binding_set_auth_info()
rpc_server_register_auth_info().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_inq_if_ids — returns a vector of interface identifiers of interfaces a server offers

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_inq_if_ids(
        rpc_binding_handle_t binding,
        rpc_if_id_vector_t **if_id_vector,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a binding handle. To receive interface identifiers from a remote application, the calling application specifies a server binding handle for that application. To receive interface information about itself, the application specifies NULL.

If the binding handle supplied refers to partially bound binding information and the binding information contains a nil object UUID, then this routine returns the rpc_s_binding_incomplete status code. To avoid this situation, the application can obtain a fully bound server binding handle by calling the **rpc_ep_resolve_binding**( ) routine.

Output

if_id_vector Returns the address of an interface identifier vector.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_no_interfaces
No interfaces registered.

rpc_s_mgmt_op_disallowed
Not authorised for operation.

In addition to the above values, **status** can return the value of **status** from an application-defined authorisation function. The prototype for such a function is defined in the **authorization_fn** argument description in **rpc_mgmt_set_authorization_fn**( ).

DESCRIPTION

An application calls the **rpc_mgmt_inq_if_ids**( ) routine to obtain a vector of interface identifiers listing the interfaces registered by a server with the RPC run-time system.

If a server has not registered any interfaces with the run-time system, this routine returns a rpc_s_no_interfaces status code and an **if_id_vector** argument value of NULL.

The binding handle supplied in the **binding** argument must refer to binding information that is fully bound or contains a non-nil object UUID. If the binding handle supplied refers to partially bound binding information that contains a nil object UUID, the routine returns the rpc_s_binding_incomplete status code.

The RPC run-time system allocates memory for the interface identifier vector. The application calls

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

the **rpc_if_id_vector_free**( ) routine to release the memory used by this vector.

By default, the RPC run-time system allows all clients to remotely call this routine. To restrict remote calls of this routine, a server application supplies an authorisation function using the **rpc_mgmt_set_authorization_fn**( ) routine.

RETURN VALUE

None.

SEE ALSO

rpc_ep_resolve_binding( )
rpc_if_id_vector_free( )
rpc_mgmt_set_authorization_fn( )
rpc_server_register_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_inq_server_princ_name — returns a server's principal name

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_inq_server_princ_name(
       rpc_binding_handle_t binding,
       unsigned32 authn_svc,
       unsigned_char_t **server_princ_name,
       unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a server binding handle for the server from which **server_princ_name** is returned. A server application can supply the value NULL to return its own principal name.

If the binding handle supplied refers to partially bound binding information and the binding information contains a nil object UUID, this routine fails with the rpc_s_binding_incomplete status code. Applications can avoid this situation by calling the **rpc_ep_resolve_binding**( ) routine to obtain a fully bound server binding handle.

authn_svc Specifies the authentication service for which a principal name is returned. (See Annex D for possible values of this argument.)

Output

server_princ_name Returns a principal name. This name is registered for the authentication service in the **authn_svc** argument by the server referenced in **binding**. If the server registered multiple principal names, only one of them is returned.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_no_princ_name
No principal name registered.

rpc_s_not_authorized
Not authorised for operation.

rpc_s_unknown_authn_service
Unknown authentication service.

In addition to the above values, **status** can return the value of **status** from an application-defined authorisation function. The prototype for such a function is defined in the **authorization_fn** argument description in **rpc_mgmt_set_authorization_fn**( ).

DESCRIPTION

An application calls routine **rpc_mgmt_inq_server_princ_name**( ) to obtain the principal name of a server that is registered for a specified authentication service.

The RPC run-time system allocates memory for the string returned in the **server_princ_name**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

argument. The application should call the **rpc_string_free**( ) routine to deallocate that memory.

By default, the RPC run-time system allows all clients to call this routine remotely. To establish non-default authorisation for this or other management calls, a server application supplies an authorisation function by calling the **rpc_mgmt_set_authorization_fn**( ) routine.

RETURN VALUE

None.

SEE ALSO

rpc_binding_inq_object( )
rpc_binding_set_auth_info( )
rpc_ep_resolve_binding( )
rpc_mgmt_set_authorization_fn( )
rpc_server_register_auth_info( )
rpc_string_free( )
uuid_is_nil( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_inq_stats — returns RPC run-time statistics

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_inq_stats(
        rpc_binding_handle_t binding,
        rpc_stats_vector_t **statistics,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a server binding handle. To receive statistics about a remote application, the calling application specifies a server binding handle for that application. To receive statistics itself, the application specifies NULL. To avoid this situation, applications can obtain a fully bound server binding handle by calling routine **rpc_ep_resolve_binding**().

Output

statistics Returns the statistics vector for the server specified by the **binding** argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_mgmt_op_disallowed
Not authorised for operation.

In addition to the above values, **status** can return the value of **status** from an application-defined authorisation function. The prototype for such a function is defined in the **authorization_fn** argument description in **rpc_mgmt_set_authorization_fn**().

DESCRIPTION

The **rpc_mgmt_inq_stats**() routine returns statistics from the RPC run-time system about a specified server. The statistics returned refer to all calls on the server by all clients.

The elements of a statistics vector are described in Subclause 3.1.

The binding handle supplied in the **binding** argument must refer to binding information that is fully bound or contains a non-nil object UUID. If the binding handle supplied refers to partially bound binding information that contains a nil object UUID, the routine returns the rpc_s_binding_incomplete status code.

The RPC run-time system allocates memory for the statistics vector. The application calls the **rpc_mgmt_stats_vector_free**() routine to release the memory that the statistics vector used.

By default, the RPC run-time system allows all clients to remotely call this routine. To restrict remote calls of this routine, a server application supplies an authorisation function using the **rpc_mgmt_set_authorization_fn**() routine.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RETURN VALUE

None.

SEE ALSO

rpc_ep_resolve_binding()
rpc_mgmt_set_authorization_fn()
rpc_mgmt_stats_vector_free().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_is_server_listening — tells whether a server is listening for remote procedure calls

SYNOPSIS

```
#include <dce/rpc.h>

boolean32 rpc_mgmt_is_server_listening(
      rpc_binding_handle_t binding,
      unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a server binding handle. To determine if a remote server is listening for remote procedure calls, the application specifies a server binding handle for that server. To determine if the application itself is listening for remote procedure calls, the application specifies NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_comm_failure
Communications failure.

rpc_s_mgmt_op_disallowed
Not authorised for operation.

rpc_s_binding_incomplete
Binding lacks both an object UUID and an endpoint.

In addition to the above values, **status** can return the value of **status** from an application-defined authorisation function. The prototype for such a function is defined in the **authorization_fn** argument description in **rpc_mgmt_set_authorization_fn**().

DESCRIPTION

The **rpc_mgmt_is_server_listening**() routine determines whether the server specified in the **binding** argument is listening for remote procedure calls.

This routine returns a value of TRUE if the server has called the **rpc_server_listen**() routine.

RETURN VALUE

Returns one of the Boolean values TRUE or FALSE.

The following table gives the interpretation of each possible combination of return value and **status** value.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Value Returned	Status Code	Explanation
TRUE	rpc_s_ok	The specified server is listening for remote procedure calls.
FALSE	rpc_s_ok or rpc_s_comm_failure	The specified server is not listening for remote procedure calls, or the server could not be reached.
FALSE	rpc_s_mgmt_op_disallowed	Not authorised for operation.

SEE ALSO

rpc_server_listen()
rpc_mgmt_set_authorization_fn()
rpc_ep_resolve_binding().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_set_authorization_fn — establishes an authorisation function for processing remote calls to a server's management routines

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_set_authorization_fn(
        rpc_mgmt_authorization_fn_t authorization_fn,
        unsigned32 *status);
```

ARGUMENTS

Input

authorization_fn Specifies an authorisation function. The RPC server run-time system automatically calls this function whenever the server run-time system receives a client request to execute one of the remote management routines. The server must implement this function.

Applications specify NULL to unregister a previously registered authorisation function. After such a call, default authorisations are used.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

Server applications call **rpc_mgmt_set_authorization_fn**() to establish an authorisation function that controls access to the server's remote management routines. (See Section 2 for an explanation of how remote management routines are implemented in servers.)

When a server has not called **rpc_mgmt_set_authorization_fn**(), or calls with a NULL value for **authorization_fn**, the server run-time system uses the default authorisations shown in the following table.

Remote Routine	Default Authorisation
rpc_mgmt_inq_if_ids()	Enabled
rpc_mgmt_inq_server_princ_name()	Enabled
rpc_mgmt_inq_stats()	Enabled
rpc_mgmt_is_server_listening()	Enabled
rpc_mgmt_stop_server_listening()	Disabled

In the table, "Enabled" indicates that all clients are allowed to execute the remote routine, and "Disabled" indicates that all clients are prevented from executing the remote routine.

A server calls **rpc_mgmt_set_authorization_fn**() to establish non-default authorisations.

The following C definition for rpc_mgmt_authorization_fn_t() shows the prototype for the authorisation function that the server must implement:

```
typedef boolean32 (*rpc_mgmt_authorization_fn_t)
  (
   rpc_binding_handle_t client_binding,          /* in  */
   unsigned32           requested_mgmt_operation, /* in  */
   unsigned32           *status                   /* out */
  );
```

When a client requests one of the server's remote management functions, the server run-time system calls the authorisation function with two arguments: **client_binding** and **requested_mgmt_operation**. The authorisation function uses these arguments to determine

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

whether the calling client is allowed to execute the requested management routine.

The **requested_mgmt_operation** value depends on the remote management routine requested, as shown in the following table.

Called Remote Routine	**requested_mgmt_operation** Value
rpc_mgmt_inq_if_ids()	rpc_c_mgmt_inq_if_ids
rpc_mgmt_inq_server_princ_name()	rpc_c_mgmt_inq_princ_name
rpc_mgmt_inq_stats()	rpc_c_mgmt_inq_stats
rpc_mgmt_is_server_listening()	rpc_c_mgmt_is_server_listen
rpc_mgmt_stop_server_listening()	rpc_c_mgmt_stop_server_listen

The authorisation function must handle all of these values.

The authorisation function returns a Boolean value to indicate whether the calling client is allowed access to the requested management function. If the authorisation function returns TRUE, the management routine is allowed to execute. If the authorisation function returns FALSE, the management routine does not execute. In the latter case, the management routine returns a **status** value to the client that depends on the **status** value returned by the authorisation function:

- If the **status** value returned by the authorisation function is either 0 (zero) or rpc_s_ok, then the **status** value rpc_s_mgmt_op_disallowed is returned to the client by the remote management routine.
- If the authorisation function returns any other **status** value, that **status** value is returned to the client by the remote management routine.

The server must implement the authorisation function in a thread-safe manner.

RETURN VALUE

None.

SEE ALSO

rpc_mgmt_ep_unregister()
rpc_mgmt_inq_if_ids()
rpc_mgmt_inq_server_princ_name()
rpc_mgmt_inq_stats()
rpc_mgmt_is_server_listening()
rpc_mgmt_stop_server_listening().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_set_cancel_timeout — sets the lower bound on the time to wait before timing out after forwarding a cancel

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_set_cancel_timeout(
        signed32 seconds,
        unsigned32 *status);
```

ARGUMENTS

Input

seconds An integer specifying the number of seconds to wait for a server to acknowledge a cancel. To specify that a client waits an infinite amount of time, supply the value rpc_c_cancel_infinite_timeout.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_mgmt_set_cancel_timeout**( ) routine resets the amount of time the RPC run-time system waits for a server to acknowledge a cancel before orphaning the call.

The application specifies either to wait forever or to wait a length of time specified in seconds. If the value of **seconds** is 0 (zero), the remote procedure call is immediately orphaned when the RPC run time detects and forwards a pending cancel; control returns immediately to the client application. The default value is rpc_c_cancel_infinite_timeout, which specifies waiting forever for the call to complete.

The value for the cancel timeout applies to all remote procedure calls made in the current thread. A multi-threaded client that wishes to change the timeout value must call this routine in each thread of execution.

RETURN VALUE

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_set_com_timeout — sets the communication timeout value in a binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_set_com_timeout(
        rpc_binding_handle_t binding,
        unsigned32 timeout,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies the server binding handle whose timeout value is set.

timeout Specifies a communications timeout value.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_mgmt_set_com_timeout**() routine resets the communications timeout value in a server binding handle. The **timeout** argument specifies the relative amount of time to spend trying to communicate with the server. Depending on the protocol sequence for the specified binding handle, the **timeout** argument acts only as advice to the RPC run-time system.

After the initial relationship is established, subsequent communications for the binding handle can revert to not less than the default timeouts for the protocol service. This means that after setting a short initial timeout for establishing a connection, calls in progress are not timed out any sooner than the default.

The timeout value can be any of the following:

rpc_c_binding_min_timeout
: Attempts to communicate for the minimum amount of time for the network protocol being used. This value favours response time over correctness in determining whether the server is running.

rpc_c_binding_default_timeout
: Attempts to communicate for an average amount of time for the network protocol being used. This value gives equal consideration to response time and correctness in determining whether a server is running. This is the default value.

rpc_c_binding_max_timeout
: Attempts to communicate for the longest finite amount of time for the network protocol being used. This value favours correctness in determining whether a server is running over response time.

rpc_c_binding_infinite_timeout
: Attempts to communicate forever.

Note that these values represent relative, rather than absolute, values.

RETURN VALUE

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

SEE ALSO

rpc_mgmt_inq_com_timeout().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_mgmt_set_server_stack_size — specifies the stack size for server call threads

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_set_server_stack_size(
       unsigned32 thread_stack_size,
       unsigned32 *status);
```

ARGUMENTS

Input

thread_stack_size Specifies the stack size, in bytes, for call threads created when the server calls **rpc_server_listen**(). Select this value based on the stack requirements of the remote procedures offered by the server.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_not_supported
Not supported.

DESCRIPTION

The **rpc_mgmt_set_server_stack_size**() routine specifies the thread stack size to use when the RPC run-time system creates call threads for executing remote procedure calls. Call threads are created when the server applications calls **rpc_server_listen**(). The **max_calls_exec** argument to the **rpc_server_listen**() routine specifies the number of call threads created.

The server must call this routine before calling the **rpc_server_listen**() routine. If a server does not call this routine, the default per-thread stack size from the underlying threads package is used.

The thread stack size set by **rpc_mgmt_set_server_stack_size**() applies only to call threads created when the server subsequently calls **rpc_server_listen**().

Some thread packages do not support the specification or modification of thread stack sizes.

RETURN VALUE

None.

SEE ALSO

rpc_server_listen().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_stats_vector_free — frees a statistics vector

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_stats_vector_free(
        rpc_stats_vector_t **stats_vector,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

stats_vector Specifies a statistics vector. On successful return, **stats_vector** contains the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

An application calls the **rpc_mgmt_stats_vector_free**() routine to release the memory used to store a vector of statistics obtained with a call to **rpc_mgmt_inq_stats**().

RETURN VALUE

None.

SEE ALSO

rpc_mgmt_inq_stats().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_mgmt_stop_server_listening — tells a server to stop listening for remote procedure calls

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_mgmt_stop_server_listening(
        rpc_binding_handle_t binding,
        unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a server binding handle for the server that is to stop listening for remote procedure calls. Specifying NULL causes the application itself to stop listening.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_comm_failure
Communications failure.

rpc_s_not_authorized
Not authorised for operation.

rpc_s_unknown_if
Server does not support this interface.

rpc_s_binding_incomplete
Binding lacks both an object UUID and an endpoint.

In addition to the above values, **status** can return the value of **status** from an application-defined authorisation function. The prototype for such a function is defined in the **authorization_fn** argument description in **rpc_mgmt_set_authorization_fn**().

DESCRIPTION

The **rpc_mgmt_stop_server_listening**() routine directs a server to stop listening for remote procedure calls.

On receipt of such a request, the RPC run-time system stops accepting new remote procedure calls.

RETURN VALUE

None.

SEE ALSO

rpc_server_listen()
rpc_mgmt_set_authorization_fn()
rpc_ep_resolve_binding().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_network_inq_protseqs — returns all protocol sequences supported both by the local implementation of the RPC run-time system and the operating system

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_network_inq_protseqs(
        rpc_protseq_vector_t **protseq_vector,
        unsigned32 *status);
```

ARGUMENTS

Output

protseq_vector Returns the address of a protocol sequence vector.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success. One or more protocol sequences are supported by the local implementation of the RPC run-time system and the operating system.

rpc_s_no_protseqs
No supported protocol sequences.

DESCRIPTION

The **rpc_network_inq_protseqs**() routine obtains a vector containing the protocol sequences supported by the RPC run-time system and the operating system. A protocol sequence is supported when the RPC run-time system and the operating system implement the protocol stack specified by the protocol sequence.

In order to offer its services remotely, a server must accept remote procedure calls over one or more of the supported protocol sequences. When there are no supported protocol sequences, this routine returns the rpc_s_no_protseqs status code and the value NULL in the returned **protseq_vector**.

The application is responsible for calling the **rpc_protseq_vector_free**() routine to release the memory used by the returned protocol sequence vector.

RETURN VALUE

None.

SEE ALSO

rpc_protseq_vector_free().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_network_is_protseq_valid — tells whether the specified protocol sequence is valid and/or is supported by the local implementation of the RPC run-time system and the operating system

SYNOPSIS

```
#include <dce/rpc.h>

boolean32 rpc_network_is_protseq_valid(
        unsigned_char_t *protseq,
        unsigned32 *status);
```

ARGUMENTS

Input

protseq Specifies a protocol sequence. Annex B lists valid protocol sequence identifiers that may be used for this argument.

The **rpc_network_is_protseq_valid**( ) routine determines whether this argument contains a valid and/or supported protocol sequence.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success. The protocol sequence is valid and supported by the local implementation of the RPC run-time system and the operating system.

rpc_s_invalid_protseq
Invalid protocol sequence.

rpc_s_protseq_not_supported
The protocol sequence is valid but not supported by the local implementation of the RPC run-time system and/or the operating system.

DESCRIPTION

The **rpc_network_is_protseq_valid**( ) routine determines whether a specified protocol sequence is both valid and supported and thus available for making remote procedure calls.

- A protocol sequence is valid if it is one of the protocol sequence strings recognised by the implementation. Information about valid protocol sequence strings is given in Annex B.
- A protocol sequence is supported if the local RPC run-time system and the operating system implement the protocol stack specified by the protocol sequence.

An application can obtain the set of valid and supported protocol sequences by calling the **rpc_network_inq_protseqs**( ) routine.

RETURN VALUE

The **rpc_network_is_protseq_valid**( ) routine returns the following values:

TRUE The protocol sequence specified in the **protseq** argument is valid and supported by the RPC run-time system and the operating system. The routine also returns the status code rpc_s_ok in the **status** argument.

FALSE The protocol sequence specified in the **protseq** argument is not valid or not supported. The routine also returns a status code not equal to rpc_s_ok.

SEE ALSO

rpc_network_inq_protseqs( )
rpc_string_binding_parse( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_export — establishes a name service database entry with binding handles and/or object UUIDs for a server

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_export(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. See Annex C for the possible values of this argument.

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry to which binding handles and/or object UUIDs are exported. The entry name syntax is identified by the argument **entry_name_syntax**.

if_handle Identifies the interface to export. Specifying the value NULL indicates that there are no binding handles to export, and the **binding_vec** argument is ignored.

binding_vec Specifies a vector of server bindings to export. The application specifies the value NULL for this argument when there are no binding handles to export.

object_uuid_vec Identifies a vector of object UUIDs offered by the application. The application constructs this vector. NULL indicates that there are no object UUIDs to export.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_binding_export**( ) routine allows a server application to make bindings to an interface it offers available in a name service. A server application can also use this routine to make available the object UUIDs of application resources.

To export an interface, the server application calls **rpc_ns_binding_export**( ) with an interface and server binding handles that reference bindings a client can use to access the server.

A server can export interfaces and objects in a single call to this routine, or it can export them

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

separately.

If the entry in the name service database specified by the **entry_name** argument does not exist, the **rpc_ns_binding_export**( ) routine tries to create it. In this case a server must have the correct permissions to create the entry.

Before calling the **rpc_ns_binding_export**( ) routine to export interfaces (but not to export object UUIDs), a server must do the following:

- Register one or more protocol sequences with the local RPC run-time system by calling the one of the following routines:

  rpc_server_use_protseq( )
  rpc_server_use_protseq_if( )
  rpc_server_use_protseq_ep( )
  rpc_server_use_all_protseqs( )
  rpc_server_use_all_protseqs( )

- Obtain a list of server bindings by calling the **rpc_server_inq_bindings**( ) routine.

  The application uses the vector returned from the **rpc_server_inq_bindings**( ) routine to supply the **binding_vec** argument for **rpc_ns_binding_export**( ). To prevent a binding from being exported, the application can set the selected vector element to the value NULL.

In addition to calling **rpc_ns_binding_export**( ), a server that calls either of the routines **rpc_server_use_all_protseqs**( ) or **rpc_server_use_protseq**( ) must also register with the local endpoint map by calling the **rpc_ep_register**( ) or **rpc_ep_register_no_replace**( ) routines.

If a server exports an interface to the same entry in the name service database more than once, the second and subsequent calls to this routine add the binding information and object UUIDs only if they differ from the ones in the server entry. Existing data is not removed from the entry.

Permissions Required

The application needs both read permission and write permission to the target name service entry. If the entry does not exist, the application also needs insert permission to the parent directory.

RETURN VALUE

None.

SEE ALSO

rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_ns_binding_unexport( )
rpc_ns_mgmt_binding_unexport( )
rpc_ns_mgmt_entry_create( )
rpc_server_inq_bindings( )
rpc_server_use_all_protseqs( )
rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )
rpc_server_use_protseq_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_import_begin — creates an import context for an interface and an object in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_import_begin(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        uuid_t *obj_uuid,
        rpc_ns_handle_t *import_context,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. See Annex C for the possible values of this argument.

The value rpc_c_ns_syntax_default specifies the syntax given by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry where the search for compatible binding handles begins. The entry name syntax is identified by the argument **entry_name_syntax**.

To use the entry name found in the **RPC_DEFAULT_ENTRY** environment variable, the application supplies NULL or an empty string ("") for this argument. When the default entry name is used, the RPC run-time system uses the default name syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

if_handle Specifies the interface to import.

If the interface specification has not been exported or is of no concern to the caller, the application specifies NULL for this argument. In this case the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and, depending on the value of argument **obj_uuid**, contain the specified object UUID.

obj_uuid Specifies an object UUID.

If the application specifies a nil UUID for this argument, and the compatible server exported object UUIDs, bindings returned by subsequent calls to **rpc_ns_binding_import_done**() contain one of the exported object UUIDs. If the server did not export any object UUIDs, the returned binding handles contain a nil object UUID.

If the application specifies a non-nil UUID for this argument, subsequent calls to **rpc_ns_binding_import_done**() return bindings that contain the specified non-nil object UUID.

Output

import_context Returns a name service handle for use with the **rpc_ns_binding_import_done**() and **rpc_ns_binding_import_done**() routines.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

rpc_s_ok Success.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_binding_import_begin**( ) routine creates an import context for importing compatible server bindings. Compatible bindings are those that offer the interface and object UUIDS specified by the **if_handle** and **obj_uuid** arguments.

The application must call this routine to create an import context before calling the **rpc_ns_binding_import_done**( ) routine.

After importing bindings, the the application calls the **rpc_ns_binding_import_done**( ) routine to delete the import context.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_import_done( )
rpc_ns_binding_import_done( )
rpc_ns_mgmt_handle_set_exp_age( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_import_done — deletes the import context for searching the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_import_done(
        rpc_ns_handle_t *import_context,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

import_context Specifies the name service handle to delete. (A name service handle is created by calling the **rpc_ns_binding_import_begin**( ) routine.)

On success, returns the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_binding_import_done**( ) routine deletes an import context created by calling the **rpc_ns_binding_import_begin**( ) routine. This deletion does not affect any previously imported bindings.

Note: Typically, a client calls this routine after completing remote procedure calls to a server using a binding handle returned from the **rpc_ns_binding_import_done**( ) routine. A client program calls this routine for each created import context, regardless of the status returned from the **rpc_ns_binding_import_done**( ) routine, or the success in making remote procedure calls.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_import_begin( )
rpc_ns_binding_import_done( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_import_done — returns a binding handle of a compatible server (if found) from the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_import_next(
        rpc_ns_handle_t import_context,
        rpc_binding_handle_t *binding,
        unsigned32 *status);
```

ARGUMENTS

Input

import_context Specifies a name service handle. Applications obtain this handle by calling **rpc_ns_binding_import_begin**( ).

Output

binding Returns a compatible server binding handle.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_bindings
No more bindings.

rpc_s_no_ns_permission
No permission for name service operation.

DESCRIPTION

The **rpc_ns_binding_import_done**( ) routine returns one compatible, exported server binding handle. Compatible binding handles are specified by the **import_context** argument that the application obtains by calling **rpc_ns_binding_import_begin**( ). (See **rpc_ns_binding_import_begin**( ) for further information on the selection of compatible binding handles.)

Note: A similar routine is **rpc_ns_binding_lookup_next**( ), which returns a vector of compatible server binding handles for one or more servers.

On successive calls, this routine returns a series of compatible bindings, one at a time. Successive invocations eventually return all such bindings from all relevant entries. When there are no further compatible bindings, the routine returns a status code of rpc_s_no_more_bindings and the value NULL in the **binding** argument.

The **rpc_ns_binding_import_done**( ) routine obeys the binding search rules specified in Section 2 and Subclause 2.4. The order in which bindings are returned to the application depends on the search rules in the following way: when the search encounters a binding attribute containing compatible bindings, successive calls to **rpc_ns_binding_import_done**( ) return all compatible bindings from that attribute in random order.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Notes: Bindings are returned from each binding attribute in random order in order to provide load balancing among bindings.

Implementations may buffer bindings from each binding attribute in an implementation-dependent sized buffer. If the number of compatible bindings from a binding attribute exceeds the buffer size, bindings are returned from the buffer in random order until the buffer is exhausted. Then the buffer is refilled from the same binding attribute. This process is repeated until all the bindings from the binding attribute have been returned. In this case, returned bindings are randomised within a buffer, but not among buffers.

Because of this randomisation, the order in which bindings are returned can be different for each new search beginning with a call to **rpc_ns_binding_import_done**( ). This means that the order in which bindings are returned to an application can be different each time the application is run.

The returned compatible binding contains an object UUID. Its value depends on the value of the **obj_uuid** argument to the **rpc_ns_binding_import_begin**( ) call that returned **import_context**:

- When **obj_uuid** contains a non-nil object UUID, the returned binding contains that object UUID.
- When **obj_uuid** contains a nil object UUID, the object UUID returned in the binding depends on how the servers exported object UUIDs to namespace entries. For a given namespace entry in the traversal path:
  - — When servers did not export any object UUIDs to the given entry, the returned binding contains a nil object UUID.
  - — When servers exported one object UUID to the given entry, the returned binding contains that object UUID.
  - — When servers exported multiple object UUIDs to the given entry, the returned binding contains one of the object UUIDs. **rpc_ns_binding_import_done**( ) selects the returned object UUID in an unspecified way.

The client application can use the returned compatible binding handle to make a remote procedure calls to the server.

Note: If the client fails to communicate with the server, it can call **rpc_ns_binding_import_done**( ) again.

Each time the client calls the **rpc_ns_binding_import_done**( ) routine, the routine returns another server binding handle. Different binding handles can refer to different protocol sequences from the same server.

If the same compatible binding is encountered more than once in a search, **rpc_ns_binding_import_done**( ) may choose not to return every instance of the binding. The **rpc_ns_binding_import_done**( ) routine allocates memory for the returned **binding** argument. When a client application finishes with the binding handle, it must call the **rpc_binding_free**( ) routine to deallocate the memory. Each call to the **rpc_ns_binding_import_done**( ) routine requires a corresponding call to the **rpc_binding_free**( ) routine.

The application calls the **rpc_ns_binding_import_done**( ) routine when it has finished using the import context. This deletes the import context.

Permissions Required

The application needs read permission to the starting name service entry and to any object entry in the resulting traversal path.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_import_begin( )
rpc_ns_binding_import_done( )
rpc_ns_binding_inq_entry_name( )
rpc_ns_binding_lookup_begin( )
rpc_ns_binding_lookup_done( )

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_ns_binding_lookup_next()
rpc_ns_binding_select()
rpc_ns_binding_export()
rpc_ns_mgmt_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_binding_inq_entry_name — returns the name of the name service database entry that contains a given binding handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_inq_entry_name(
       rpc_binding_handle_t binding,
       unsigned32 entry_name_syntax,
       unsigned_char_t **entry_name,
       unsigned32 *status);
```

ARGUMENTS

Input

binding Specifies a server binding handle whose entry name in the name service database is returned.

entry_name_syntax An integer value that specifies the syntax of the returned **entry_name**. (See Annex C for information about values of this argument.)

To use the syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, the application provides the value rpc_c_ns_syntax_default.

Output

entry_name Returns the name of the entry in the name service database in which **binding** was found. The returned name conforms to the specified syntax.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_no_entry_name
No entry name for binding.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_binding_inq_entry_name**() routine returns the name of the name service database entry that contains a binding handle for a compatible server.

The RPC run-time system allocates memory for the string returned in **entry_name**. The application calls the **rpc_string_free**() routine to deallocate this memory.

The **binding** argument must come from a call to one of the **rpc_ns_binding_import_done**(), rpc_ns_binding_lookup_next() or **rpc_ns_binding_select**() routines.

When the binding handle specified in the **binding** argument is not from an entry in the name service database, this routine returns the rpc_s_no_entry_name status code.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_binding_from_string_binding()
rpc_ns_binding_import_done()

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_ns_binding_lookup_next()
rpc_ns_binding_select()
rpc_string_free().

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_binding_lookup_begin — creates a lookup context for an interface and an object in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_lookup_begin(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        uuid_t *object_uuid,
        unsigned32 binding_max_count,
        rpc_ns_handle_t *lookup_context,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax given by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry where the search for compatible binding handles begins. The entry name syntax is identified by the argument **entry_name_syntax**.

To use the entry name found in the **RPC_DEFAULT_ENTRY** environment variable, the application supplies NULL or an empty string ("") for this argument. When the default entry name is used, the RPC run-time system uses the default name syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

if_handle Specifies the interface to import.

If the interface specification has not been exported or is of no concern to the caller, the application specifies NULL for this argument. In this case the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and, depending on the value of argument **obj_uuid**, contain the specified object UUID.

obj_uuid Specifies an object UUID.

If the application specifies a nil UUID for this argument, and the compatible server exported object UUIDs, binding handles returned by subsequent calls to **rpc_ns_binding_lookup_next**() contain one of the exported object UUIDs. If the server did not export any object UUIDs, the returned binding handles contain a nil object UUID.

If the application specifies a non-nil UUID for this argument, subsequent calls to **rpc_ns_binding_lookup_next**() return binding handles that contain the specified non-nil object UUID.

binding_max_count Sets the maximum number of bindings to return in the **binding_vector** argument of the **rpc_ns_binding_lookup_next**() routine.

To use the default count, specify rpc_c_binding_max_count.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Output

lookup_context Returns the name service handle for use with the **rpc_ns_binding_lookup_next**( ) and rpc_ns_binding_lookup_done( ) routines.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_binding_lookup_begin**( ) routine creates a lookup context for locating compatible server binding handles for servers. Compatible binding handles are those that offer the interface and object UUIDS specified by the **if_handle** and **obj_uuid** arguments.

The application must call this routine to create a lookup context before calling the **rpc_ns_binding_lookup_next**( ) routine.

After looking up binding handles, the the application calls the **rpc_ns_binding_lookup_done**( ) routine to delete the import context.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_lookup_next( )
rpc_ns_binding_lookup_done( )
rpc_ns_mgmt_handle_set_exp_age( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_lookup_done — deletes the lookup context for searching the name service database (used by client applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_lookup_done(
        rpc_ns_handle_t *lookup_context,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

lookup_context Specifies the name service handle to delete. (A name service handle is created by calling the **rpc_ns_binding_lookup_begin**( ) routine.)

On success, returns the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_binding_lookup_done**( ) routine deletes a lookup context created by calling the **rpc_ns_binding_lookup_begin**( ) routine.

Note: Typically, a client calls this routine after completing remote procedure calls to a server using a binding handle returned from the **rpc_ns_binding_lookup_next**( ) routine. A client program calls this routine for each created lookup context, regardless of the status returned from the **rpc_ns_binding_lookup_next**( ) routine, or success in making remote procedure calls.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_lookup_begin( )
rpc_ns_binding_lookup_next( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_lookup_next — returns a list of binding handles of one or more compatible servers, if found, from the name service database (used by client applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_lookup_next(
        rpc_ns_handle_t lookup_context,
        rpc_binding_vector_t **binding_vec,
        unsigned32 *status);
```

ARGUMENTS

Input

lookup_context Specifies a name service handle. This handle is returned from the **rpc_ns_binding_lookup_begin**() routine.

Output

binding_vec Returns a vector of compatible server binding handles.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_bindings
No more bindings.

rpc_s_no_ns_permission
No permission for name service operation.

DESCRIPTION

The **rpc_ns_binding_lookup_next**() routine returns a vector of compatible exported server binding handles. Compatible binding handles are specified by the **import_context** argument that the application obtains by calling **rpc_ns_binding_lookup_begin**(). (See **rpc_ns_binding_lookup_begin**() for further information on the selection of compatible binding handles.)

A similar routine is **rpc_ns_binding_import_done**(), which returns **one** compatible server binding handle.

On successive calls, this routine traverses entries in the name service database, returning compatible server binding handles from each entry. The routine can return multiple binding handles from each entry. Successive invocations eventually return all such binding handles from all relevant entries. When there are no further compatible binding handles, the routine returns a status code of rpc_s_no_more_bindings and the value NULL in **binding_vec**.

The **rpc_ns_binding_lookup_next**() routine obeys the binding search rules specified in Section 2 and Subclause 2.4.

Each returned compatible binding handle contains an object UUID. Its value depends on the value

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

of the **obj_uuid** argument to the **rpc_ns_binding_lookup_begin**() call that returned **lookup_context**:

- When **obj_uuid** contains a non-nil object UUID, the returned binding handle contains that object UUID.
- When **obj_uuid** contains a nil object UUID, the object UUID returned in the binding handle depends on how the servers exported object UUIDs to namespace entries. For a given namespace entry in the traversal path:
  - — When servers did not export any object UUIDs to the given entry, the returned binding handle contains a nil object UUID.
  - — When servers exported one object UUID to the given entry, the returned binding handle contains that object UUID.
  - — When servers exported multiple object UUIDs to the given entry, the returned binding handle contains one of the object UUIDs. **rpc_ns_binding_lookup_next**() selects the returned object UUID in an unspecified way.

Notes: From the returned vector of server binding handles, the client application can employ its own criteria for selecting individual binding handles, or the application can call the **rpc_ns_binding_select**() routine to select a binding handle. The **rpc_binding_to_string_binding**() and **rpc_string_binding_parse**() routines are useful for a client creating its own selection criteria.

The client application can use the selected binding handle to attempt a remote procedure call to the server. If the client fails to communicate with the server, it can select another binding handle from the vector. When all of the binding handles in the vector are used, the client application calls the **rpc_ns_binding_lookup_next**() routine again.

Each time the client calls the **rpc_ns_binding_lookup_next**() routine, the routine returns another vector of binding handles. The binding handles returned in each vector are randomly ordered. The vectors returned from multiple calls to this routine are also randomly ordered.

When looking up compatible binding handles from a profile, the binding handles from entries of equal profile priority are randomly ordered in the returned vector. In addition, the vector returned from a call to **rpc_ns_binding_lookup_next**() contains only compatible binding handles from entries of equal profile priority. This means the returned vector may be partially full.

For example, if the **binding_max_count** argument value in **rpc_ns_binding_lookup_begin**() was 5 and **rpc_ns_binding_lookup_next**() finds only three compatible binding handles from profile entries of priority 1, **rpc_ns_binding_lookup_next**() returns a partially full binding vector (with three binding handles). The next call to **rpc_ns_binding_lookup_next**() creates a new binding vector and begins looking for compatible binding handles from profile entries of priority 0.

If the same compatible binding is encountered more than once in a search, **rpc_ns_binding_lookup_next**() may choose not to return every instance of the binding.

When the search finishes, the routine returns a status code of rpc_s_no_more_bindings and returns the value NULL in **binding_vec**.

Note: The **rpc_ns_binding_inq_entry_name**() routine is called by an application in order to obtain the name of the entry in the name service database where the binding handle came from.

The **rpc_ns_binding_lookup_next**() routine allocates memory for the returned **binding_vec**. When an application finishes with the vector, it must call the **rpc_binding_vector_free**() routine to deallocate the memory. Each call to the **rpc_ns_binding_lookup_next**() routine requires a corresponding call to the **rpc_binding_vector_free**() routine.

The application calls the **rpc_ns_binding_lookup_done**() to delete the lookup context when it is done with a search or to begin a new search for compatible servers (by calling the **rpc_ns_binding_lookup_begin**() routine). The order of binding handles returned can be different for each new search. This means that the order in which binding handles are returned to an application can be different each time the application is run.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application needs read permission to the specified name service object entry (the starting name service entry) and to any name service object entry in the resulting search path.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_import_done()
rpc_ns_binding_lookup_begin()
rpc_ns_binding_lookup_done()
rpc_ns_binding_select()
rpc_binding_vector_free()
rpc_ns_binding_inq_entry_name()
rpc_binding_to_string_binding()
rpc_string_binding_parse()
rpc_ns_mgmt_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_select — returns a binding handle from a list of compatible binding handles

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_select(
        rpc_binding_vector_t *binding_vec,
        rpc_binding_handle_t *binding,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

binding_vec Specifies the vector of compatible server binding handles from which a binding handle is selected. The returned binding vector no longer references the selected binding handle (which is returned separately in **binding**).

Output

binding Returns a selected server binding handle.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_no_more_bindings
No more bindings.

DESCRIPTION

The **rpc_ns_binding_select**() routine randomly chooses and returns a server binding handle from a vector of server binding handles.

Each time the application calls the **rpc_ns_binding_select**() routine, the routine returns another binding handle from the vector.

When all of the binding handles are returned from the vector, the routine returns a status code of rpc_s_no_more_bindings and returns the value NULL in **binding**.

The RPC run-time system allocates storage for the data referenced by the returned **binding**. When an application finishes with the binding handle, it calls the **rpc_binding_free**() routine to deallocate the storage. Each call to the **rpc_ns_binding_select**() routine requires a corresponding call to the **rpc_binding_free**() routine.

Note: Instead of using this routine, applications can select a binding handle according to their specific needs. In this case the **rpc_binding_to_string_binding**() and **rpc_string_binding_parse**() routines are useful to the applications since the routines work together to extract the individual fields of a binding handle for examination.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_binding_free()
rpc_binding_to_string_binding()
rpc_ns_binding_lookup_next()
rpc_string_binding_parse().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_binding_unexport — removes binding handles and/or object UUIDs from an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_binding_unexport(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry from which binding handles or objects UUIDs are removed. The entry name syntax is identified by the argument **entry_name_syntax**.

if_handle An interface specification for the binding handles to be removed from the name service database. The value NULL indicates that no binding handles are removed.

object_uuid_vec A vector of object UUIDs to be removed from the name service database. The application constructs this vector. The value NULL indicates that no object UUIDs are removed.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_interface_not_found
Interface not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_all_objs_unexported
Not all objects unexported.

rpc_s_not_rpc_entry
Not an RPC entry.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

DESCRIPTION

The **rpc_ns_binding_unexport**( ) routine allows an application to unexport (that is, remove) one of the following from an entry in the name service database:

- all the binding handles for an interface
- one or more object UUIDs for a resource or resources
- both binding handles and object UUIDs.

The **rpc_ns_binding_unexport**( ) routine removes only those binding handles that match the interface UUID and the major and minor interface version numbers found in the **if_handle** argument. To remove multiple versions of an interface, applications use the **rpc_ns_mgmt_binding_unexport**( ) routine.

Note: A server application can remove an interface and objects in a single call to this routine, or it can remove them separately.

If the **rpc_ns_binding_unexport**( ) routine does not find any binding handles for the specified interface, the routine returns an rpc_s_interface_not_found status code and does not remove the object UUIDs, if any are specified.

If the application specifies both binding handles and object UUIDs, the object UUIDs are removed only if the **rpc_ns_binding_unexport**( ) routine succeeds in removing the binding handles.

If any of the specified object UUIDs are not found, routine **rpc_ns_binding_unexport**( ) returns the status code rpc_s_not_all_objs_unexported.

Notes: Besides calling this routine, an application also calls the **rpc_ep_unregister**( ) routine to unregister any endpoints that the server previously registered with the local endpoint map.

Applications normally call this routine only when a server is expected to be unavailable for an extended time.

Permissions Required

The application needs both read permission and write permission to the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_ep_unregister( )
rpc_ns_binding_export( )
rpc_ns_mgmt_binding_unexport( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_entry_expand_name — returns a canonicalised version of an entry name

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_entry_expand_name(
       unsigned32 entry_name_syntax,
       unsigned_char_t *entry_name,
       unsigned_char_t **expanded_name,
       unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax Specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

An application can supply the value rpc_c_ns_syntax_default to use the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry to canonicalise. The entry name syntax is identified by the argument **entry_name_syntax**.

Output

expanded_name Returns a pointer to the canonicalised version of argument **entry_name**.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

An application calls the **rpc_ns_entry_expand_name**() routine to obtain a canonicalised version of an entry name. Canonicalisation rules depend on the underlying name service.

The RPC run-time system allocates memory for the returned **expanded_name**. The application is responsible for calling the **rpc_string_free**() routine to free this memory.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_string_free().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_entry_inq_resolution — resolves the cell namespace components of a name and returns partial results.

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_entry_inq_resolution(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned_char_t **resolved_name,
        unsigned_char_t **unresolved_name,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable. The value rpc_c_ns_syntax_default corresponds to the name syntax specified in Part 1 of this International Standard.

entry_name The composite entry name on which the attempted name resolution is to be done.

Input/Output

resolved_name Returns a pointer to the resolved portion of argument **entry_name**. The resolved name is null terminated and does not contain trailing component separators (that is, it contains no trailing "/" (slash) characters.

resolved_name Returns a pointer to the unresolved portion of argument **entry_name**. The unresolved name is a relative name not containing leading component separators (that is, it contains no leading "/" (slash) characters).

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include (refer to AES/DC Volume RPC, Annex E for encodings):

rpc_s_ok Success.

rpc_s_partial_results The entry name was only partially resolved within the cell namespace and the value of **unresolved_name** points to the residual of the name.

rpc_s_unsupported_name_syntax The requested name syntax is not supported.

DESCRIPTION

The **rpc_ns_entry_inq_resolution**() routine attempts to read a name entry in the cell namespace. If the read was successful then the full resolution of the entry name is returned. If the read was not successful due to the entry not being found in the cell namespace, then the partial result of the name resolution is returned, with status code set to rpc_s_partial_results.

If the status code is rpc_s_partial_results and the retuen parameter **resolved_name** is non-empty, the contents of **resolved_name** can be used in subsequent calls to the NSI interface to obtain the binding handle for the name server that controls the namespace in which the entry specified by the value of **unresolved_name** lies.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application requires read permission for the name entries that are resolved in the cell namespace.

RETURN VALUE

None.

RELATED INFORMATION

Functions:
rpc_string_free( ).

CDS remote operations:
cds_Enumerate*( ),
cds_ReadAttribute*( ) IDL Operations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_entry_object_inq_begin — creates an inquiry context for viewing the objects of an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_entry_object_inq_begin(
      unsigned32 entry_name_syntax,
      unsigned_char_t *entry_name,
      rpc_ns_handle_t *inquiry_context,
      unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry in the name service database for which object UUIDs are viewed. The entry name syntax is identified by the argument **entry_name_syntax**.

Output

inquiry_context Returns an inquiry context for use with routines **rpc_ns_entry_object_inq_next**() and **rpc_ns_entry_object_inq_done**().

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_entry_object_inq_begin**() routine creates an inquiry context for viewing the object UUIDs exported to **entry_name**.

Before calling the **rpc_ns_entry_object_inq_next**() routine, the application must first call this routine to create an inquiry context.

When finished viewing the object UUIDs, the application calls the **rpc_ns_entry_object_inq_done**() routine to delete the inquiry context.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_export()
rpc_ns_entry_object_inq_done()
rpc_ns_entry_object_inq_next()
rpc_ns_mgmt_handle_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_entry_object_inq_done — deletes the inquiry context for viewing the objects of an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_entry_object_inq_done(
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

inquiry_context Specifies the inquiry context to delete. (An inquiry context is created by calling the **rpc_ns_entry_object_inq_begin**( ) routine.)

On success, returns the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_entry_object_inq_done**( ) routine deletes an inquiry context created by calling the **rpc_ns_entry_object_inq_begin**( ) routine.

An application calls this routine after viewing exported object UUIDs using the **rpc_ns_entry_object_inq_next**( ) routine.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_entry_object_inq_begin( )
rpc_ns_entry_object_inq_next( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_entry_object_inq_next — returns one object at a time from an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_entry_object_inq_next(
        rpc_ns_handle_t inquiry_context,
        uuid_t *obj_uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

inquiry_context Specifies an inquiry context. The application obtains the inquiry context by calling the **rpc_ns_entry_object_inq_begin**( ) routine.

Output

obj_uuid Returns an exported object UUID.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_members
No more members.

rpc_s_no_ns_permission
No permission for name service operation.

DESCRIPTION

The **rpc_ns_entry_object_inq_next**( ) routine returns one of the object UUIDs exported to an entry in the name service database. The **entry_name** argument in the **rpc_ns_entry_object_inq_begin**( ) routine specifies the entry.

An application can view all of the exported object UUIDs by repeatedly calling the **rpc_ns_entry_object_inq_next**( ) routine. When all the object UUIDs are viewed, this routine returns an rpc_s_no_more_members status. The returned object UUIDs are returned in unspecified order.

The application supplies the memory for the object UUID returned in **obj_uuid**.

After viewing the object UUIDs, the application must call the **rpc_ns_entry_object_inq_done**( ) routine to delete the inquiry context.

The order in which routine **rpc_ns_entry_object_inq_next**( ) returns object UUIDs can be different for each viewing of an entry. This means that the order in which an application receives object UUIDs can be different each time the application is run.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application needs read permission for the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_export()
rpc_ns_entry_object_inq_begin()
rpc_ns_entry_object_inq_done()
rpc_ns_mgmt_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_group_delete — deletes a group attribute

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_group_delete(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 *status);
```

ARGUMENTS

Input

group_name_syntax An integer value that specifies the syntax of argument **group_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

group_name The name of the group to delete. The group name syntax is identified by the argument **group_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_group_delete**() routine deletes the group attribute from the specified entry in the name service database.

Neither the specified entry nor the entries represented by the group members are deleted.

Permissions Required

The application needs write permission to the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_group_mbr_add()
rpc_ns_group_delete().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_group_mbr_add — adds an entry name to a group; if necessary, creates the entry

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_add(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 *status);
```

ARGUMENTS

Input

group_name_syntax An integer value that specifies the syntax of argument **group_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

group_name The name of the group to which the member is added. The group name syntax is identified by the argument **group_name_syntax**.

member_name_syntax An integer value that specifies the syntax of argument **member_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

member_name The name of the group member to add. The member name syntax is identified by the argument **member_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_group_mbr_add**() routine adds a group member to the group attribute of a name service entry. The **group_name** argument specifies the entry.

If the specified **group_name** entry does not exist, this routine creates the entry with a group attribute and adds the group member specified by the **member_name** argument. In this case, the application must have permission to create the entry.

An application can add the entry in argument **member_name** to a group before it creates the member itself.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application needs both read permission and write permission for the target name service entry. If the entry does not exist, the application also needs insert permission for the parent directory.

RETURN VALUE

None.

SEE ALSO

rpc_group_mbr_remove()
rpc_ns_mgmt_entry_create().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_group_mbr_inq_begin — creates an inquiry context for viewing group members

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_inq_begin(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 member_name_syntax,
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input

group_name_syntax An integer value that specifies the syntax of argument **group_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

group_name The name of the group to view. The group name syntax is identified by the argument **group_name_syntax**.

member_name_syntax An integer value that specifies the syntax of return argument **member_name** for the **rpc_ns_group_mbr_inq_next**( ) routine. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

Output

inquiry_context Returns an inquiry context for use with the **rpc_ns_group_mbr_inq_next**( ) and **rpc_ns_group_mbr_inq_done**( ) routines.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_group_mbr_inq_begin**( ) routine creates an inquiry context for viewing the members of an RPC group.

The application calls this routine to create an inquiry context before calling the **rpc_ns_group_mbr_inq_next**( ) routine.

When finished viewing the RPC group members, the application calls the **rpc_ns_group_mbr_inq_done**( ) routine to delete the inquiry context.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_group_mbr_add()
rpc_ns_group_mbr_inq_done()
rpc_ns_group_mbr_inq_next()
rpc_ns_mgmt_handle_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_group_mbr_inq_done — deletes the inquiry context for a group

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_inq_done(
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

inquiry_context Specifies the inquiry context to delete. (An inquiry context is created by calling the **rpc_ns_group_mbr_inq_begin**( ) routine.)

On success, returns the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_group_mbr_inq_done**( ) routine deletes an inquiry context created by calling the **rpc_ns_group_mbr_inq_begin**( ) routine.

An application calls this routine after viewing RPC group members using the **rpc_ns_group_mbr_inq_next**( ) routine.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_group_mbr_inq_begin( )
rpc_ns_group_mbr_inq_next( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_group_mbr_inq_next — returns one member name at a time from a group

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_inq_next(
        rpc_ns_handle_t inquiry_context,
        unsigned_char_t **member_name,
        unsigned32 *status);
```

ARGUMENTS

Input

inquiry_context Specifies an inquiry context. The application obtains the inquiry context by calling the **rpc_ns_group_mbr_inq_begin**( ) routine.

Output

member_name Returns a pointer to an RPC group member name.

The syntax of the **member_name** argument depends on the value of **inquiry_context**. The application specifies this syntax with the **member_name_syntax** argument when it calls **rpc_ns_group_mbr_inq_begin**( ) to obtain the inquiry context.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_members
No more members.

rpc_s_no_ns_permission
No permission for name service operation.

DESCRIPTION

The **rpc_ns_group_mbr_inq_next**( ) routine returns one member of the RPC group specified by the **group_name** argument in the **rpc_ns_group_mbr_inq_begin**( ) routine.

An application can view all the members of an RPC group by repeatedly calling the **rpc_ns_group_mbr_inq_next**( ) routine. When all the group members have been viewed, this routine returns an rpc_s_no_more_members status. The group members are returned in unspecified order.

On each call to this routine that returns a member name, the RPC run-time system allocates memory for the returned **member_name**. The application calls the **rpc_string_free**( ) routine for each returned **member_name** string.

After viewing the RPC group's members, the application must call the **rpc_ns_group_mbr_inq_done**( ) routine to delete the inquiry context.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application needs read permission to the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_group_mbr_inq_begin()
rpc_ns_group_mbr_inq_done()
rpc_string_free()
rpc_ns_mgmt_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_group_mbr_remove — removes an entry name from a group (used by client, server or management applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_remove(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 *status);
```

ARGUMENTS

Input

group_name_syntax An integer value that specifies the syntax of argument **group_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

group_name The name of the group from which the member is removed. The group name syntax is identified by the argument **group_name_syntax**.

member_name_syntax An integer value that specifies the syntax of argument **member_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

member_name The name of the group member to remove. The member name syntax is identified by the argument **member_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_group_member_not_found
Group member not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_group_mbr_remove**() routine removes a member from the group attribute in the **group_name** entry.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application needs both read permission and write permission for the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_group_mbr_add().

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

NAME

rpc_ns_mgmt_binding_unexport — removes multiple binding handles, or object UUIDs, from an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_binding_unexport(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_id_t *if_id,
        unsigned32 vers_option,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry from which binding handles or object UUIDs are removed. The entry name syntax is identified by the argument **entry_name_syntax**.

if_id Specifies an interface identifier for the binding handles to be removed from the name service database. The value NULL indicates that no binding handles are removed.

vers_option Specifies how the **rpc_ns_mgmt_binding_unexport**() routine uses the **vers_major** and the **vers_minor** fields of the **if_id** argument.

The following list presents the accepted values for this argument:

Value	Description
rpc_c_vers_all	Unexports (that is, removes) all bindings for the interface UUID in **if_id**, regardless of the version numbers.
rpc_c_vers_compatible	Removes those bindings for the interface UUID in **if_id** with the same major version as in **if_id**, and with a minor version greater than or equal to the minor version in **if_id**.
rpc_c_vers_exact	Removes those bindings for the interface UUID in **if_id** with the same major and minor versions as in **if_id**.
rpc_c_vers_major_only	Removes those bindings for the interface UUID in **if_id** with the same major version as in **if_id** (ignores the minor version).
rpc_c_vers_upto	Removes those bindings that offer a version of the specified interface UUID less than or equal to the specified major and minor version.

object_uuid_vec A vector of object UUIDs to be removed from the name service database. The application constructs this vector. The value NULL indicates that no object UUIDs are removed

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_interface_not_found
Interface not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_all_objs_unexported
Not all objects unexported.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_mgmt_binding_unexport**() routine allows an application to unexport (that is, remove) one of the following from an entry in the name service database:

- all the binding handles for a specified interface UUID, qualified by the interface version numbers (major and minor)
- one or more object UUIDs for a resource or resources
- both binding handles and object UUIDs.

An application can remove an interface and objects in a single call to this routine, or it can remove them separately.

If the **rpc_ns_mgmt_binding_unexport**() routine does not find any binding handles for the specified interface, the routine returns an rpc_s_interface_not_found status and does not remove the object UUIDs, if any are specified.

If the application specifies both binding handles and object UUIDs, the object UUIDs are removed only if the routine succeeds in removing the binding handles.

If any of the specified object UUIDs are not found, routine **rpc_ns_mgmt_binding_unexport**() returns the rpc_not_all_objs_unexported status code.

Notes: Besides calling this routine, an application also calls the **rpc_mgmt_ep_unregister**() routine to remove any servers that have registered with the local endpoint map.

Applications normally call this routine only when a server is expected to be unavailable for an extended time.

Permissions Required

The application needs both read permission and write permission to the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_mgmt_ep_unregister()
rpc_ns_binding_export()

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_ns_binding_unexport( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_mgmt_entry_create — creates an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_entry_create(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry to create. The entry name syntax is identified by the argument **entry_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_already_exists
Name service entry already exists.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_mgmt_entry_create**() routine creates an entry in the name service database.

A management application can call **rpc_ns_mgmt_entry_create**() to create an entry in the name service database for use by another application that does not itself have the necessary name service permissions to create an entry.

Permissions Required

The application that calls **rpc_ns_mgmt_entry_create**() needs insert permission for the parent directory. In order to modify the entry, the application for which it was created needs both read permission and write permission.

RETURN VALUE

None.

SEE ALSO

rpc_ns_mgmt_entry_delete()

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_mgmt_entry_delete — deletes an entry from the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_entry_delete(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax An integer value that specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name The name of the entry to delete. The entry name syntax is identified by the argument **entry_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_mgmt_entry_delete**() routine removes an RPC entry from the name service database.

Note: Management applications use this routine only when an entry is no longer needed, such as when a server is permanently removed from service. If the entry is a member of a group or profile, it must also be deleted from the group or profile.

Permissions Required

The application needs read permission for the target name service entry. The application also needs delete permission for the entry or for the parent directory.

RETURN VALUE

None.

SEE ALSO

rpc_ns_mgmt_entry_create().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_mgmt_entry_inq_if_ids — returns the list of interface IDs exported to an entry in the name service database

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_entry_inq_if_ids(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_id_vector_t **if_id_vec,
        unsigned32 *status);
```

ARGUMENTS

Input

entry_name_syntax Specifies the syntax of argument **entry_name**. (See Annex C for the possible values of this argument.)

An application can supply the value rpc_c_ns_syntax_default to use the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

entry_name Specifies the entry in the name service database for which an interface identifier vector is returned. The entry name must conform to the syntax specified by **entry_name_syntax**.

Output

if_id_vec Returns the interface identifier vector.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_interfaces_exported
No interfaces were exported to the entry.

rpc_s_no_ns_permission
No permission for name service operation.

DESCRIPTION

The **rpc_ns_mgmt_entry_inq_if_ids**() routine returns an interface identifier vector that contains interface IDs from the binding information in a name service entry. This routine returns binding information from the specified entry only; it does not search any profile or group members contained in the specified entry.

In implementations that cache name service data, this routine always gets its returned data directly from the name service, updating any local cache.

Applications must call **rpc_if_id_vector_free**() to free the memory used by the returned **if_id_vec**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Permissions Required

The application needs read permission to the target name service entry.

RETURN VALUE

None.

SEE ALSO

rpc_if_id_vector_free()
rpc_if_inq_id()
rpc_ns_binding_export().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_mgmt_handle_set_exp_age — sets the expiration age for cached copies of name service data obtained with a given handle

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_handle_set_exp_age(
        rpc_ns_handle_t ns_handle,
        unsigned32 expiration_age,
        unsigned32 *status);
```

ARGUMENTS

Input

ns_handle Specifies the name service handle for which the application supplies an expiration age.

expiration_age Specifies the expiration age, in seconds, for cached copies of name service data obtained with **ns_handle**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_mgmt_handle_set_exp_age**() routine sets the expiration age for the specified name service handle, **ns_handle**. This expiration age is used, instead of the application's global expiration age, for all name service operations obtained using **ns_handle**. Expiration age is further described in **rpc_ns_mgmt_inq_exp_age**().

Because name service caching is implementation-dependent, the effect of setting a handle's expiration age (on subsequent name service operations performed with the handle) is implementation dependent.

Note: In implementations that perform name service caching, setting the handle expiration age to a small value may cause operations that retrieve data from the name service to update cached data frequently. An expiration age of 0 (zero) forces an update on each operation involving the same attribute data. Frequent updates may adversely affect the performance both of the calling application and any other applications that share the same cache.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_binding_import_begin()
rpc_ns_binding_lookup_begin()
rpc_ns_entry_object_inq_begin()
rpc_ns_group_mbr_inq_begin()
rpc_ns_mgmt_inq_exp_age()
rpc_ns_mgmt_set_exp_age()
rpc_ns_profile_elt_inq_begin().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_mgmt_inq_exp_age — returns the application's global expiration age for cached copies of name service data

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_inq_exp_age(
        unsigned32 *expiration_age,
        unsigned32 *status);
```

ARGUMENTS

Input

None.

Output

expiration_age The application's global expiration age, in seconds, for cached copies of name service data.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_mgmt_inq_exp_age**() routine returns the application's global name service expiration age.

The effect of expiration age on name service operations is implementation-dependent. For implementations that cache, the expiration age is the maximum amount of time, in seconds, that a cached copy of data from a name service attribute is considered valid by name service operations that read data from a name service. Name service routines that may be affected by expiration age are as follows:

rpc_ns_binding_import_done()
rpc_ns_binding_lookup_next()
rpc_ns_entry_object_inq_next()
rpc_ns_group_mbr_inq_next()
rpc_ns_profile_elt_inq_next()

Implementations that cache look for cached copies of the requested data. When there is no cached copy, the operation creates one with fresh data from the name service database. When there is a cached copy, the operation compares its age with the calling application's expiration age. If the copy's age exceeds the expiration age, the operation attempts to update the cached copy with fresh data from the name service. If updating fails, the cached data remains unchanged and the requested operation fails, returning the rpc_s_name_service_unavailable status code.

Implementations that do not cache behave as if the expiration age were 0 (zero). Fresh data is always retrieved from the name service.

Every application maintains a global expiration age value. When an application begins running, the RPC run-time system specifies an implementation-dependent default global expiration age for the application. Applications may change this value by calling **rpc_ns_mgmt_set_exp_age**(). Applications may also set the expiration ages of individual name service handles. Whenever a name service operation is performed using a handle for which the application has not set an expiration age, the global expiration age value is used.

Permissions Required

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

RETURN VALUE

None.

SEE ALSO

rpc_ns_mgmt_handle_set_exp_age()
rpc_ns_mgmt_set_exp_age()
rpc_ns_binding_import_done()
rpc_ns_binding_lookup_next()
rpc_ns_entry_object_inq_next()
rpc_ns_group_mbr_inq_next()
rpc_ns_profile_elt_inq_next().

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_mgmt_set_exp_age — Modifies an application's global expiration age for cached copies of name service data

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_mgmt_set_exp_age(
        unsigned32 expiration_age,
        unsigned32 *status);
```

ARGUMENTS

Input

expiration_age Specifies the application's global expiration age, in seconds, for cached copies of name service data.

Applications can reset the expiration age to the implementation-specific default by supplying the value rpc_c_ns_default_exp_age.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_mgmt_set_exp_age**() routine sets the application's global name service expiration age.

The effect of expiration age on name service operations is implementation-dependent. For implementations that cache name service data, the expiration age is the maximum amount of time, in seconds, that a cached copy of data from a name service attribute is considered valid by name service operations that read data from a name service. Name service routines that may be affected by expiration age are as follows:

rpc_ns_binding_import_done()
rpc_ns_binding_lookup_next()
rpc_ns_entry_object_inq_next()
rpc_ns_group_mbr_inq_next()
rpc_ns_profile_elt_inq_next()

Implementations that cache look for cached copies of the requested data. When there is no cached copy, the operation creates one with fresh data from the name service database. When there is a cached copy, the operation compares its age with the calling application's expiration age. If the copy's age exceeds the expiration age, the operation attempts to update the cached copy with fresh data from the name service. If updating fails, the cached data remains unchanged and the requested operation fails, returning the rpc_s_name_service_unavailable status code.

Implementations that do not cache behave as if the expiration age were 0 (zero). Fresh data is always retrieved from the name service.

Every application maintains a global expiration age value. When an application begins running, the RPC run-time system specifies an implementation-dependent default global expiration age for the application. Applications may query this value by calling **rpc_ns_mgmt_inq_exp_age**(). Applications may also set the expiration ages of individual name service handles. Whenever a name service operation is performed using a handle for which the application has not set an expiration age, the global expiration age value is used.

Note: In implementations that cache name service data, setting the expiration age to a small value may cause operations that retrieve data from the name service to update cached data frequently. An expiration age of 0 (zero) forces an update on each operation involving the same attribute data. Frequent updates may adversely affect the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

performance both of the calling application and any other applications that share the same cache.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_mgmt_handle_set_exp_age()
rpc_ns_mgmt_set_exp_age()
rpc_ns_binding_import_done()
rpc_ns_binding_lookup_next()
rpc_ns_entry_object_inq_next()
rpc_ns_group_mbr_inq_next()
rpc_ns_profile_elt_inq_next().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_profile_delete — deletes a profile attribute

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_profile_delete(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        unsigned32 *status);
```

ARGUMENTS

Input

profile_name_syntax An integer value that specifies the syntax of argument **profile_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

profile_name The name of the profile to delete. The profile name syntax is identified by the argument **profile_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_profile_delete**() routine deletes the profile attribute from the specified entry in the name service database.

Neither the specified entry nor the entry names included as members in each profile element are deleted.

Note: Use this routine cautiously; deleting a profile may break a hierarchy of profiles.

Permissions Required

The application needs write permission to the target name service profile entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_profile_elt_add()
rpc_ns_profile_elt_remove().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_profile_elt_add — adds an element to a profile; if necessary, creates the entry

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_add(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        rpc_if_id_t *if_id,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 priority,
        unsigned_char_t *annotation,
        unsigned32 *status);
```

ARGUMENTS

Input

profile_name_syntax An integer value that specifies the syntax of argument **profile_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

profile_name Specifies the RPC profile that receives the new element. The profile name syntax is identified by the argument **profile_name_syntax**.

if_id Specifies the interface identifier of the new profile element. To add or replace the default profile element, specify NULL.

member_name_syntax An integer value that specifies the syntax of argument **member_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

member_name Specifies an entry in the name service database to include in the new profile element. The member name syntax is identified by the argument **member_name_syntax**.

priority An integer value (0 to 7) that specifies the relative priority for using the new profile element during the import and lookup operations. A value of 0 (zero) is the highest priority. A value of 7 is the lowest priority. Two or more elements can have the same priority.

The default profile element has a priority of 0. When adding the default profile, the result is unspecified if the application specifies a value other than 0 here.

annotation Specifies an annotation string that is stored as part of the new profile element. The string can be up to rpc_c_annotation_max characters long, including the null terminator. The application specifies NULL or the empty string ("") if there is no annotation string.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_profile_elt_add**() routine adds an element to the profile attribute of the entry in the name service database specified by the **profile_name** argument.

If the **profile_name** entry does not exist, this routine creates the entry with a profile attribute and adds the profile element specified by the **if_id**, **member_name**, **priority** and **annotation** arguments. In this case, the application must have permission to create the entry.

If an element with the specified member name and interface identifier is already in the profile, this routine updates the element's priority and annotation string using the values provided in the **priority** and **annotation** arguments.

An application can add the entry in argument **member_name** to a profile before it creates the entry itself.

Permissions Required

The application needs both read permission and write permission for the target name service profile entry. If the entry does not exist, the application also needs insert permission for the parent directory.

RETURN VALUE

None.

SEE ALSO

rpc_if_inq_id()
rpc_ns_mgmt_entry_create()
rpc_ns_profile_elt_remove().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_ns_profile_elt_inq_begin — creates an inquiry context for viewing the elements in a profile

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_inq_begin(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        unsigned32 inquiry_type,
        rpc_if_id_t *if_id,
        unsigned32 vers_option,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input

profile_name_syntax An integer value that specifies the syntax of argument **profile_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

profile_name Specifies the RPC profile to view. The profile name syntax is identified by the argument **profile_name_syntax**.

inquiry_type An integer value that specifies the type of inquiry to perform on the profile. The following list describes the valid values for this argument:

Value	Description
rpc_c_profile_default_elt	Searches the profile for the default profile element, if any. The **if_id**, **vers_option**, and **member_name** arguments are ignored.
rpc_c_profile_all_elts	Returns every element from the profile. The **if_id**, **vers_option**, and **member_name** arguments are ignored.
rpc_c_profile_match_by_if	Searches the profile for those elements that contain the interface identifier specified by the **if_id** and **vers_option** values. The **member_name** argument is ignored.
rpc_c_profile_match_by_mbr	Searches the profile for those elements that contain the member name specified by the **member_name** argument. The **if_id** and **vers_option** arguments are ignored.
rpc_c_profile_match_by_both	Searches the profile for those elements that contain the interface identifier and member name specified by the **if_id, vers_option** and **member_name** arguments.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

if_id Specifies the interface identifier of the profile elements to be returned by the **rpc_ns_profile_elt_inq_next**( ) routine.

This argument is meaningful only when specifying a value of rpc_c_profile_match_by_if or rpc_c_profile_match_by_both for the **inquiry_type** argument. Otherwise, this argument is ignored and the application can specify the value NULL.

vers_option Specifies how the **rpc_ns_profile_elt_inq_next**( ) routine uses the **if_id** argument.

This argument is used only when specifying a value of rpc_c_profile_match_by_if or rpc_c_profile_match_by_both for the **inquiry_type** argument. Otherwise, this argument is ignored.

The following list describes the valid values for this argument:

Value	Description
rpc_c_vers_all	Returns profile elements that offer the specified interface UUID, regardless of the version numbers.
rpc_c_vers_compatible	Returns profile elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.
rpc_c_vers_exact	Returns profile elements that offer the specified version of the specified interface UUID.
rpc_c_vers_major_only	Returns profile elements that offer the same major version of the specified interface UUID (ignores the minor version).
rpc_c_vers_upto	Returns profile elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version.

member_name_syntax An integer value that specifies the syntax of argument **member_name** in this routine and the syntax of argument **member_name** in the **rpc_ns_profile_elt_inq_next**( ) routine. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

member_name Specifies the member name that the **rpc_ns_profile_elt_inq_next**( ) routine looks for in profile elements. The member name syntax is identified by the argument **member_name_syntax**.

This argument is meaningful only when specifying a value of rpc_c_profile_match_by_mbr or rpc_c_profile_match_by_both for the **inquiry_type** argument. Otherwise, this argument is ignored.

Output

inquiry_context Returns a name service handle for use with the **rpc_ns_profile_elt_inq_next**( ) and **rpc_ns_profile_elt_inq_done**( ) routines.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

rpc_s_ok Success.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_profile_elt_inq_begin**( ) routine creates an inquiry context for viewing the elements in a profile.

Using the **inquiry_type** and **vers_option** arguments, an application specifies which of the following profile elements will be returned from calls to the **rpc_ns_profile_elt_inq_next**( ) routine:

- the default element
- all elements
- those elements with the specified interface identifier
- those elements with the specified member name
- those elements with both the specified interface identifier and member name.

The application calls this routine to create an inquiry context before calling the **rpc_ns_profile_elt_inq_next**( ) routine.

When finished viewing profile elements, the application calls the **rpc_ns_profile_elt_inq_done**( ) routine to delete the inquiry context.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_if_inq_id( )
rpc_ns_mgmt_handle_set_exp_age( )
rpc_ns_profile_elt_inq_done( )
rpc_ns_profile_elt_inq_next( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_profile_elt_inq_done — deletes the inquiry context for a profile

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_inq_done(
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

inquiry_context Specifies the name service handle to delete. (A name service handle is created by calling the **rpc_ns_profile_elt_inq_begin**() routine.)

On success, returns the value NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_ns_profile_elt_inq_done**() routine deletes an inquiry context created by calling the **rpc_ns_profile_elt_inq_begin**() routine.

An application calls this routine after viewing profile elements using the **rpc_ns_profile_elt_inq_next**() routine.

Permissions Required

None.

RETURN VALUE

None.

SEE ALSO

rpc_ns_profile_elt_inq_begin()
rpc_ns_profile_elt_inq_next().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_profile_elt_inq_next — returns one element at a time from a profile

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_inq_next(
        rpc_ns_handle_t inquiry_context,
        rpc_if_id_t *if_id,
        unsigned_char_t **member_name,
        unsigned32 *priority,
        unsigned_char_t **annotation,
        unsigned32 *status);
```

ARGUMENTS

Input

inquiry_context Specifies a name service handle. This handle is returned from the **rpc_ns_profile_elt_inq_begin**( ) routine.

Output

if_id Returns the interface identifier of the profile element.

member_name Returns a pointer to the profile element's member name.

The syntax of the returned name is specified by the **member_name_syntax** argument in the **rpc_ns_profile_elt_inq_begin**( ) routine.

Specifying NULL prevents the routine from returning this argument. In this case the application need not call the **rpc_string_free**( ) routine.

priority Returns the profile element priority.

annotation Returns the annotation string for the profile element. If there is no annotation string in the profile element, the empty string ("") is returned.

Specifying NULL prevents the routine from returning this argument. In this case the application need not call the **rpc_string_free**( ) routine.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_class_version_mismatch
Name service entry has incompatible RPC class version.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_elements
No more elements.

rpc_s_no_ns_permission
No permission for name service operation.

DESCRIPTION

The **rpc_ns_profile_elt_inq_next**( ) routine returns one element from the profile specified by the **profile_name** argument in the **rpc_ns_profile_elt_inq_begin**( ) routine.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The selection criteria for the element returned are based on the **inquiry_type** argument in routine **rpc_ns_profile_elt_inq_begin**(). Routine **rpc_ns_profile_elt_inq_next**() returns all the components (interface identifier, member name, priority, annotation string) of a profile element.

An application can view all the selected profile entries by repeatedly calling the **rpc_ns_profile_elt_inq_next**() routine. When all the elements have been viewed, this routine returns an rpc_s_no_more_elements status code. The returned elements are unordered.

On each call to this routine that returns a profile element, the RPC run-time system allocates memory for the returned **member_name** and **annotation** strings. The application is responsible for calling the **rpc_string_free**() routine for each returned **member_name** and **annotation** string.

After viewing the profile's elements, the application must call the **rpc_ns_profile_elt_inq_done**() routine to delete the inquiry context.

Permissions Required

The application needs read permission to the the target name service profile entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_profile_elt_inq_begin()
rpc_ns_profile_elt_inq_done()
rpc_string_free()
rpc_ns_mgmt_set_exp_age().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_ns_profile_elt_remove — removes an element from a profile

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_remove(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        rpc_if_id_t *if_id,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 *status);
```

ARGUMENTS

Input

profile_name_syntax An integer value that specifies the syntax of argument **profile_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

profile_name The name of the profile from which an element is removed. The profile name syntax is identified by the argument **profile_name_syntax**.

if_id Specifies the interface identifier of the profile element to be removed.

The application specifies NULL to remove the default profile member.

member_name_syntax An integer value that specifies the syntax of argument **member_name**. (See Annex C for the possible values of this argument.)

The value rpc_c_ns_syntax_default specifies the syntax specified by the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

member_name Specifies the name service entry to remove from the profile. The member name syntax is identified by the argument **member_name_syntax**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_profile_element_not_found
Profile element not found.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

DESCRIPTION

The **rpc_ns_profile_elt_remove**() routine removes a profile element from the profile attribute in the **profile_name** entry. Note that the **member_name** argument and the **if_id** argument must match exactly for an element to be removed.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The entry (**member_name**) referred to as a member in the profile element is not deleted.

Note: Use this routine cautiously. Removing elements from a profile may break a hierarchy of profiles.

Permissions Required

The application needs both read permission and write permission to the target name service profile entry.

RETURN VALUE

None.

SEE ALSO

rpc_ns_profile_delete()
rpc_ns_profile_elt_add().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_object_inq_type — returns the type of an object

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_object_inq_type(
       uuid_t *obj_uuid,
       uuid_t *type_uuid,
       unsigned32 *status);
```

ARGUMENTS

Input

obj_uuid Specifies the object UUID whose associated type UUID is returned. This may be the nil UUID.

Output

type_uuid Returns the type UUID corresponding to the object UUID supplied in argument **obj_uuid**.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_object_not_found
Object not found.

DESCRIPTION

A server application calls the **rpc_object_inq_type**( ) routine to obtain the type UUID of an object.

If the object is registered with the RPC run-time system using the **rpc_object_set_type**( ) routine, the registered type is returned.

An application can also privately maintain an object/type registration. In this case, if the application provides an object inquiry function (see **rpc_object_set_inq_fn**( )). the RPC run-time system uses that function to determine an object's type.

The following table summarises how routine **rpc_object_inq_type**( ) obtains the returned type UUID.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table 3-2 Rules for Returning an Object's Type

Has the application registered an:		Return Value
Object UUID?	Object inquiry function?	
Yes	(Ignored)	Returns the object's registered type UUID.
No	Yes	Returns the type UUID returned from calling the inquiry function.
No	No	Returns the nil UUID.

RETURN VALUE

None.

SEE ALSO

rpc_object_set_inq_fn( )
rpc_object_set_type( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_object_set_inq_fn — registers an object inquiry function

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_object_set_inq_fn(
        rpc_object_inq_fn_t inquiry_fn,
        unsigned32 *status);
```

ARGUMENTS

Input

inquiry_fn Specifies a pointer to an object type inquiry function. When an application calls the **rpc_object_inq_type**() routine, and the RPC run-time system finds that the specified object is not registered, the run-time system automatically calls this routine to determine the object's type. Specifying NULL removes a previously set inquiry function.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

A server application calls the **rpc_object_set_inq_fn**() routine to specify a function to determine an object's type. If an application privately maintains object/type registrations, the specified inquiry function returns the type UUID of an object from that registration.

The RPC run-time system automatically calls the inquiry function when the application calls routine **rpc_object_inq_type**() and the object was not previously registered by the **rpc_object_set_type**() routine. The RPC run-time system also automatically calls the inquiry function for every remote procedure call it receives if the object was not previously registered by **rpc_object_set_type**().

The following C-language definition for **rpc_object_inq_fn_t** illustrates the prototype for this function:

```
typedef void (*rpc_object_inq_fn_t)
(
    uuid_t       *object_uuid,  /* in  */
    uuid_t       *type_uuid,    /* out */
    unsigned32   *status        /* out */
```

The returned **type_uuid** and **status** values are returned as the output arguments from the **rpc_object_inq_type**() routine.

RETURN VALUE

None.

SEE ALSO

rpc_object_inq_type()
rpc_object_set_type().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_object_set_type — registers the type of an object with the RPC run-time system

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_object_set_type(
       uuid_t *obj_uuid,
       uuid_t *type_uuid,
       unsigned32 *status);
```

ARGUMENTS

Input

obj_uuid Specifies an object UUID to associate with the type UUID in the **type_uuid** argument. This may not be the nil UUID.

type_uuid Specifies the type UUID of the **obj_uuid** argument.

Specify the nil UUID to reset the object type to the default association of object UUID/nil type UUID.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_already_registered
Object already registered.

rpc_s_invalid_object
Invalid object.

DESCRIPTION

The **rpc_object_set_type**( ) routine assigns a type UUID to an object UUID.

By default, the RPC run-time system assumes that the type of all objects is nil. A server program that contains one implementation of an interface (one manager entry point vector) does not need to call this routine, provided that the server registered the interface with the nil type UUID (see **rpc_server_register_if**( ) for a description).

A server program that contains multiple implementations of an interface (multiple manager entry point vectors; that is, multiple type UUIDs) calls this routine once for each non-default object UUID the server offers. Associating each object with a type UUID tells the RPC run-time system which manager entry point vector (interface implementation) to use when the server receives a remote procedure call for a non-nil object UUID.

The RPC run-time system allows an application to set the type for an unlimited number of objects.

To remove the association between an object UUID and its type UUID (established by calling this routine), a server calls this routine again and specifies the nil UUID for the **type_uuid** argument. This resets the association between an object UUID and type UUID to the default.

A server cannot register a nil object UUID. The RPC run-time system automatically registers the nil object UUID with a nil type UUID. Attempting to set the type of a nil object UUID will result in the routine's returning the status code rpc_s_invalid_object.

Servers that want to maintain their own object UUID to type UUID mapping can use the **rpc_object_set_inq_fn**( ) routine in place of, or in addition to, the **rpc_object_set_type**( ) routine.

RETURN VALUE

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

SEE ALSO

rpc_object_set_inq_fn()
rpc_server_register_if().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_protseq_vector_free — frees the memory used by a protocol sequence vector and its protocol sequences

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_protseq_vector_free(
        rpc_protseq_vector_t **protseq_vector,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

protseq_vector Specifies the address of a pointer to a vector of protocol sequences. On return the pointer is set to NULL.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_protseq_vector_free**() routine frees the memory used to store a vector of protocol sequences when the vector was obtained by calling **rpc_network_inq_protseqs**(). Both the protocol sequences and the protocol sequence vector are freed.

RETURN VALUE

None.

SEE ALSO

rpc_network_inq_protseqs().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_server_inq_bindings — returns binding handles for communication with a server

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_inq_bindings(
        rpc_binding_vector_t **binding_vector,
        unsigned32 *status);
```

ARGUMENTS

Input

None.

Output

binding_vector Returns the address of a vector of server binding handles.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_no_bindings
No bindings.

DESCRIPTION

The **rpc_server_inq_bindings**() routine obtains a vector of server binding handles. Binding handles are created by the RPC run-time system when a server application calls any of the following routines to register protocol sequences:

rpc_server_use_all_protseqs()
rpc_server_use_all_protseqs()
rpc_server_use_protseq()
rpc_server_use_protseq_ep()
rpc_server_use_protseq_if()

The returned binding vector can contain binding handles with dynamic endpoints and binding handles with well-known endpoints, depending on which of the above routines the server application called.

A server uses the vector of binding handles for exporting to the name service, for registering with the local endpoint map, or for conversion to string bindings.

When there are no binding handles (no registered protocol sequences), this routine returns the rpc_s_no_bindings status code and returns the value NULL in **binding_vector**.

The application is responsible for calling the **rpc_binding_vector_free**() routine to deallocate the memory used by the vector.

RETURN VALUE

None.

SEE ALSO

rpc_binding_vector_free()
rpc_ep_registerP()
rpc_ep_register_no_replace()
rpc_ns_binding_export()
rpc_server_use_protseq()
rpc_server_use_all_protseqs()
rpc_server_use_protseq_ep()
rpc_server_use_protseq_if()
rpc_server_use_all_protseqs().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_inq_if — returns the manager entry point vector registered for an interface

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_inq_if(
        rpc_if_handle_t if_handle,
        uuid_t *mgr_type_uuid,
        rpc_mgr_epv_t *mgr_epv,
        unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies the interface specification whose manager entry point vector (EPV) pointer is returned in argument **mgr_epv**.

mgr_type_uuid Specifies a type UUID for the manager whose EPV pointer is returned in argument **mgr_epv**.

Specifying the nil UUID for this argument causes the routine to return a pointer to the manager EPV that is registered with **if_handle** and the nil type UUID for the manager.

Output

mgr_epv On success, returns a pointer to the manager EPV that corresponds to arguments **if_handle** and **mgr_type_uuid**.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

DESCRIPTION

An application calls the **rpc_server_inq_if**( ) routine to determine the manager EPV for a registered interface and type UUID of the manager.

RETURN VALUE

None.

SEE ALSO

rpc_server_register_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_listen — tells the RPC run-time system to listen for remote procedure calls

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_listen(
      unsigned32 max_calls_exec,
      unsigned32 *status);
```

ARGUMENTS

Input

max_calls_exec Specifies the number of concurrent executing remote procedure calls the server must be able to handle. The RPC run-time system allocates sufficient call threads to handle this number of concurrent calls.

The value rpc_c_listen_max_calls_default specifies an implementation-dependent default value ≥ 1.

Note: The five rpc_server_use_*protseq*( ) routines:

rpc_server_use_all_protseqs( )
rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )
rpc_server_use_protseq_if( ))

also specify a **max_call_requests** argument that specifies the network resources allocated for concurrent call requests. Normally the values of **max_calls_exec** and **max_call_requests** are the same. Servers are guaranteed to support the minimum of **max_calls_exec** and **max_call_requests** concurrent remote procedure calls. Applications should not rely on a server handling more than this number.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_already_listening
Server already listening.

rpc_s_max_calls_too_small
Maximum calls value is too small. Must be > 0.

rpc_s_no_protseqs_registered
No protocol sequences registered.

DESCRIPTION

The **rpc_server_listen**( ) routine causes a server to listen for remote procedure calls. The **max_calls_exec** argument specifies the number of concurrent remote procedure calls the server is guaranteed to be able to execute, assuming that the server has allocated sufficient network resources to receive this number of call requests.

A server application that specifies a value for **max_calls_exec** greater than 1 is responsible for concurrency control among the server manager routines, since each executes in a separate thread.

When the server receives more remote procedure calls than it can execute (that is, more calls than the value of **max_calls_exec**), the RPC run-time system accepts and queues additional remote procedure calls until a call execution thread is available; that is, the number of concurrently executing threads is < **max_calls_exec**. From the client's perspective a queued remote procedure call

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

appears the same as one that the server is actively executing.

The **rpc_server_listen**( ) routine returns to the caller when one of the following events occurs:

- The **rpc_mgmt_stop_server_listening**( ) routine is called by one of the server application's manager routines.
- A client makes an authorised remote **rpc_mgmt_stop_server_listening**( ) routine call to the server.

After **rpc_server_listen**( ) returns, no further calls are processed.

RETURN VALUE

None.

SEE ALSO

rpc_mgmt_stop_server_listening( )
rpc_server_register_if( )
rpc_server_use_all_protseqs( )
rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )
rpc_server_use_protseq_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_register_auth_info — registers authentication information with the RPC run-time system (used by server applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_register_auth_info(
       unsigned_char_t *server_princ_name,
       unsigned32 authn_svc,
       rpc_auth_key_retrieval_fn_t get_key_fn,
       void *arg,
       unsigned32 *status);
```

ARGUMENTS

Input

server_princ_name Specifies a server principal name to use when authenticating remote procedure calls using the service specified by **authn_svc**. The content and syntax of the name depend on the value of **authn_svc**. (See Annex D for authentication service specific syntax.)

authn_svc Specifies the authentication service to use when the server receives a remote procedure call request. (See Annex D for the possible values of this argument.)

get_key_fn Specifies the address of a server application-provided routine that returns keys suitable for the specified **authn_svc**.

To use the authentication service-specific default method of acquiring keys, NULL may be specified for this argument. (See Annex D for a description of the authentication service-specific run-time behaviour for acquiring keys.)

The following C definition for **rpc_auth_key_retrieval_fn_t** illustrates the prototype for the key acquisition routine:

```
typedef void (*rpc_auth_key_retrieval_fn_t)
 (
  void            *arg,               /* in */
  unsigned_char_t *server_princ_name, /* in */
  unsigned32      key_ver,            /* in */
  void            **key,              /* out */
  unsigned32      *status             /* out */
 );
```

The RPC run-time system passes the **server_princ_name** argument value for **rpc_server_register_auth_info**(), as the **server_princ_name** argument value for the **get_key_fn** key acquisition routine. The RPC run-time system automatically supplies a value for the **key_ver** argument.

The implementation of the key acquisition routine depends on the authentication service in use. The routine must return a key appropriate to the authentication service in the **get_key_fn** argument. For a **key_ver** value of 0 (zero), the key acquisition routine must return the most recent key available, as defined by the authentication service.

The key acquisition routine may be called from **rpc_server_register_auth_info**(). In this case, if the key acquisition routine returns a status other than rpc_s_ok, the **rpc_server_register_auth_info**() routine fails and returns the error status to the calling server.

The key acquisition routine is called by the RPC run-time system while authenticating remote procedure call requests. If it returns a status other than rpc_s_ok, the request fails and the RPC run-time system returns the error status to the calling client.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

arg Specifies an argument to pass to the key acquisition routine. (See Annex D for an explanation of how this argument is treated by the run-time system, depending on the value of **authn_svcand get_key_fn.)**

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unknown_authn_service
Unknown authentication service.

DESCRIPTION

Servers call the **rpc_server_register_auth_info**( ) routine to register an authentication service to use for authenticating remote procedure calls. A server calls this routine once for each authentication service-principal name combination that it wants to register. Severs can register a non-default key acquisition function and a key acquisition function argument when calling **rpc_server_register_auth_info**( ).

Applications may make multiple calls to **rpc_server_register_auth_info**( ) to register several principal name-authentication service combinations. When an application calls **rpc_server_register_auth_info**( ) with a combination already registered, the new registration overwrites the old one.

A client application makes authenticated remote procedure calls using a binding annotated with authentication information. If the binding has not been annotated with one of the principal name-authentication service combinations registered by the server, the client's remote procedure call request fails.

RETURN VALUE

None.

SEE ALSO

rpc_binding_set_auth_info( )
rpc_server_register_auth_info( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_register_if — registers interface/type UUID/EPV associations with the RPC run-time system

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_register_if(
        rpc_if_handle_t if_handle,
        uuid_t *mgr_type_uuid,
        rpc_mgr_epv_t mgr_epv,
        unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies the interface to register.

mgr_type_uuid Specifies a type UUID to associate with the **mgr_epv** argument. Specifying the value NULL (or a nil UUID) registers the **if_handle** with a nil type UUID.

mgr_epv Specifies the manager routine's entry point vector. Specifying NULL causes the routine to supply a default entry point vector.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_type_already_registered
An interface with the given type UUID already registered.

DESCRIPTION

The **rpc_server_register_if**() routine registers a server interface with the RPC run-time system. A server can register an unlimited number of interfaces. Once registered, an interface is available to clients through any binding handle of the server, provided that the client supports the protocols specified in the binding handle.

A server must provide the following information to register an interface with **rpc_server_register_if**():

- an interface specification; the server specifies this using the **if_handle** argument
- a type UUID and manager entry point vector (EPV) pair, using the **mgr_type_uuid** and **mgr_epv** arguments, respectively; this data pair identifies a manager to handle calls on the interface.

A server may register more than one manager per interface. To do so, the server calls **rpc_server_register_if**() at least once for each manager, specifying a different type UUID/manager EPV data pair each time.

The type UUID/manager EPV data pairs registered by this routine are used by the run-time system to determine which manager is invoked when a server receives a remote procedure call request from a client. When an RPC request is received on an interface, the RPC run-time system matches the object UUID of the call to one of the registered type UUID/manager EPV pairs and dispatches the call through the selected EPV to the appropriate manager routines.

By default, a nil object UUID matches a nil type UUID. To enable any other matches, the server must establish a mapping of object UUIDs to type UUIDs by calling the routine **rpc_object_set_type**(). The server must call **rpc_object_set_type**() at least once for each non-nil type UUID it has registered in order to make that type UUID available for dispatching calls.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Note: The mapping of object UUIDs to type UUIDs applies to all registered interfaces. If a non-nil type UUID has already been set for one interface, it is not necessary to call **rpc_object_set_type** again when that type UUID is registered for a different interface.

In an interface, one manager EPV may be registered with a nil type UUID. As the table below shows, this manager, by default, receives calls with object UUIDs that do not match another type UUID. Note that **rpc_object_set_type**( ) cannot be used to set the nil object UUID to match any other type UUID. However, a non-nil object UUID may be mapped to the nil type UUID. (See **rpc_object_set_type**( ) for further information on the object UUID to type UUID mapping.)

More than one type UUID may be registered for each manager EPV on consecutive calls to **rpc_server_register_if**( ), allowing calls whose object UUIDs match different type UUIDs to be dispatched to the same manager. However, only one manager EPV for an interface may be registered per type UUID. When an interface has been registered with a given type UUID, attempting to register it with the same type UUID results in the error rpc_s_type_already_registered.

The following table summarises the rules used by the RPC run-time system for invoking manager routines.

Object UUID of Call[1]	Has Server Set Type of Object UUID?[2]	Has Server Registered Type for Manager EPV?[3]	Dispatching Action
Nil	Not applicable[4]	Yes	Use the manager with the nil type UUID.
Nil	Not applicable[4]	No	Error: rpc_s_unknown_mgr_type. Reject the remote procedure call.
Non-nil	Yes	Yes	Use the manager with the same type UUID.
Non-nil	No	(Ignored)	Use the manager with the nil type UUID. If no manager with the nil type UUID, error: rpc_s_unknown_mgr_type. Reject the remote procedure call.
Non-nil	Yes	No	Error: rpc_s_unknown_mgr_type. Reject the remote procedure call.

a) This is the object UUID found in a binding handle for a remote procedure.

b) The server specifies the type UUID for an object by calling **rpc_object_set_type**( ).

c) The server registers the type for the manager EPV by calling **rpc_server_register_if**( ) using the same type UUID.

d) The nil object UUID is always automatically assigned the nil type UUID. It is illegal to specify a nil object UUID in the **rpc_object_set_type**( ) routine.

Specifying the Manager EPV

To use the implementation-provided default manager EPV, a server can specify the value NULL for the **mgr_epv** argument to **rpc_server_register_if**( ). A server that registers only one manager for an interface, and that wishes to use the default manager EPV needs to call **rpc_server_register_if**( ) only once, specifying the value NULL for the **mgr_epv** argument.

To use a non-default manager EPV, the server initialises a variable of the following type for each implementation of the interface:

<if-name>_v<major-version>_<minor-version>_epv_t

To register such a server supplied EPV using **rpc_server_register_if**( ), the server passes a pointer to it as the **mgr_epv** argument.

When a server registers only one manager for an interface, it can use either the default manager EPV or it can supply one. When a server registers more than one manager for an interface, it can use the default manager EPV for one of the managers, but it must supply manager EPVs for all the other managers. The server may supply manager EPVs for all managers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Specifying the Type UUID

A server may specify a nil type UUID for one of the manager EPVs registered. Calls are dispatched to a manager registered with the nil type UUID in two circumstances:

- when the call object UUID is nil
- when the call object UUID is non-nil and no type UUID has been set for the object UUID.

When a server registers only one manager for an interface, it can use either a nil or non-nil value. When a server registers more than one manager for an interface, it can use the nil type UUID for one of the manager EPVs. The server must supply distinct non-nil type UUIDs for all other manager EPVs registered. The server may supply non-nil type UUIDs for all manager EPVs registered.

The server may not specify the same type UUID for more than one manager EPV. The server may, however, specify more than one type UUID per manager EPV. To do so, the server calls **rpc_server_register_if**() more than once, each time specifying a type UUID/manager EPV pair with the same manager EPV and a different type UUID. This permits calls with object UUIDs that match different type UUIDs to be handled by the same manager.

When a server registers the nil type UUID, and does not make any calls to **rpc_object_set_type**(), all calls, regardless of object UUID, are dispatched to the manager EPV registered with the nil type UUID. In the simplest case, a server calls **rpc_server_register_if** with a NULL **mgr_epv** argument, specifying the default manager EPV, and a nil **mgr_type_uuid** argument. If such a server does not call **rpc_object_set_type**(), all calls will be dispatched to the default manager.

RETURN VALUE

None.

SEE ALSO

rpc_binding_from_string_binding()
rpc_binding_set_object()
rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ns_binding_export()
rpc_ns_binding_import_begin()
rpc_ns_binding_lookup_begin()
rpc_object_set_type()
rpc_server_unregister_if().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_unregister_if — removes an interface from the RPC run-time system

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_unregister_if(
       rpc_if_handle_t if_handle,
       uuid_t *mgr_type_uuid,
       unsigned32 *status);
```

ARGUMENTS

Input

if_handle Specifies an interface specification to unregister (that is, remove).

The application specifies NULL to remove all interfaces previously registered with the type UUID value given in the **mgr_type_uuid** argument.

mgr_type_uuid Specifies the type UUID for the manager entry point vector (EPV) to remove.

Note: This should be the same value as was provided in a call to the **rpc_server_register_if**( ) routine.

The application specifies NULL to remove the interface given in the **if_handle** argument for all previously registered type UUIDs.

The application specifies a nil UUID to remove the default manager EPV. In this case all manager EPVs registered with a non-nil type UUID remain registered.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

DESCRIPTION

The **rpc_server_unregister_if**( ) routine removes the association between an interface and a manager entry point vector (EPV).

The application specifies the manager EPV to remove by providing in the **mgr_type_uuid** argument the type UUID value specified in a call to the **rpc_server_register_if**( ) routine. Once removed, an interface is no longer available to client applications.

When an interface is removed, the RPC run-time system stops accepting new calls for that interface.

The following table summarises the behaviour of this routine.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

if_handle	*mgr_type_uuid*	**Behaviour**
non-NULL	non-NULL	Removes the manager EPV associated with the specified arguments.
non-NULL	NULL	Removes all manager EPVs associated with argument **if_handle**.
NULL	non-NULL	Removes all manager EPVs associated with argument **mgr_type_uuid**.
NULL	NULL	Removes all manager EPVs.

Note: When both of the arguments **if_handle** and **mgr_type_uuid** are given the value NULL, this call has the effect of preventing the server from receiving any new remote procedure calls since all the manager EPVs for all interfaces have been removed.

RETURN VALUE

None.

SEE ALSO

rpc_server_register_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_server_use_all_protseqs — tells the RPC run-time system to use all supported protocol sequences for receiving remote procedure calls (used by server applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_use_all_protseqs(
        unsigned32 max_call_requests,
        unsigned32 *status);
```

ARGUMENTS

Input

max_call_requests Specifies the number of concurrent remote procedure call requests that the server is guaranteed to accept from the transport. The RPC run-time system allocates sufficient network resources to handle this number of concurrent calls.

The RPC run-time system guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of **max_call_requests** and can vary for each protocol sequence.

The value rpc_c_protseq_max_reqs_default specifies an implementation dependent default value ≥1.

Note: The **rpc_server_listen** routine specifies a **max_calls_exec** argument that specifies the number of call threads the server will allocate to handle calls. Normally, the values of **max_calls_exec** and **max_call_requests** are the same. Servers are guaranteed to support the minimum of **max_calls_exec** and **max_call_requests** concurrent remote procedure calls. Applications should not rely on a server handling more than this number.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_cant_create_sock
Cannot create a transport endpoint.

rpc_s_max_descs_exceeded
Exceeded maximum number of transport endpoints.

rpc_s_no_protseqs
No supported protocol sequences.

DESCRIPTION

The **rpc_server_use_all_protseqs**() routine registers all supported protocol sequences with the RPC run-time system. A server must register at least one protocol sequence with the RPC run-time system to receive remote procedure call requests.

For each protocol sequence registered by a server, the RPC run-time system creates one or more binding handles. The RPC run-time system creates different binding handles for each protocol sequence. Each binding handle contains a dynamic endpoint that the RPC run-time system generated.

The **max_call_requests** argument allows applications to specify a maximum number of concurrent remote procedure call requests the server must handle.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

After registering protocol sequences, a server typically calls the following routines:

- **rpc_server_inq_bindings**(), which obtains a vector containing all of the server's binding handles
- **rpc_ep_register**() or rpc_ep_register_no_replace(), which register the binding handles with the local endpoint map
- **rpc_ns_binding_export**(), which places the binding handles in the name service database for access by any client
- **rpc_binding_vector_free**(), which frees the vector of server binding handles
- **rpc_server_register_if**(), which registers with the RPC run-time system those interfaces that the server offers
- **rpc_server_listen**(), which enables the reception of remote procedure calls.

To selectively register protocol sequences, a server calls the **rpc_server_use_protseq**(), **rpc_server_use_all_protseqs**(), **rpc_server_use_protseq_if**() or **rpc_server_use_protseq_ep**() routine.

RETURN VALUE

None.

SEE ALSO

rpc_binding_from_string_binding()
rpc_binding_to_string_binding()
rpc_binding_vector_free()
rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ns_binding_export()
rpc_server_inq_bindings()
rpc_server_listen()
rpc_server_register_if()
rpc_server_use_all_protseqs()
rpc_server_use_protseq()
rpc_server_use_protseq_ep()
rpc_server_use_protseq_if().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_use_all_protseqs — tells the RPC run-time system to use all the protocol sequences and endpoints specified in an interface specification for receiving remote procedure calls (used by server applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_use_all_protseqs_if(
        unsigned32 max_call_requests,
        rpc_if_handle_t if_handle,
        unsigned32 *status);
```

ARGUMENTS

Input

max_call_requests Specifies the number of concurrent remote procedure call requests that the server is guaranteed to accept from the transport. The RPC run-time system allocates sufficient network resources to handle this number of concurrent calls.

The RPC run-time system guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of **max_call_requests** and can vary for each protocol sequence.

The value rpc_c_protseq_max_reqs_default specifies an implementation-dependent default value ≥1.

Note: The **rpc_server_listen** routine specifies a **max_calls_exec** argument that specifies the number of call threads the server will allocate to handle calls. Normally, the values of **max_calls_exec** and **max_call_requests** are the same. Servers are guaranteed to support the minimum of **max_calls_exec** and **max_call_requests** concurrent remote procedure calls. Applications should not rely on a server handling more than this number.

if_handle Specifies an interface specification containing the protocol sequences and their corresponding endpoint information to use in creating binding handles. Each created binding handle contains a well-known (non-dynamic) endpoint contained in the interface specification.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_no_protseqs
No supported protocol sequences.

DESCRIPTION

The **rpc_server_use_all_protseqs**() routine registers all stub-defined protocol sequences and associated endpoint address information with the RPC run-time system. A server must register at least one protocol sequence with the RPC run-time system to receive remote procedure call requests. The **max_call_requests** argument specifies the number of concurrent remote procedure call requests the server is guaranteed to handle.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

To register selected stub-defined protocol sequences, applications use the **rpc_server_use_protseq_if**( ) routine. After calling **rpc_server_use_all_protseqs**( ), an application typically calls the following routines:

- **rpc_server_inq_bindings**( ), which obtains a vector containing all of the server's binding handles
- **rpc_ns_binding_export**( ), which places the binding handles in the name service database for access by any client
- **rpc_binding_vector_free**( ), which frees the vector of server binding handles
- **rpc_server_register_if**( ), which registers with the RPC run-time system those interfaces that the server offers
- **rpc_server_listen**( ), which enables the reception of remote procedure calls.

RETURN VALUE

None.

SEE ALSO

rpc_binding_vector_free( )
rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_ns_binding_export( )
rpc_server_inq_bindings( )
rpc_server_listen( )
rpc_server_register_if( )
rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( )
rpc_server_use_protseq_if( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_use_protseq — tells the RPC run-time system to use the specified protocol sequence for receiving remote procedure calls

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_use_protseq(
        unsigned_char_t *protseq,
        unsigned32 max_call_requests,
        unsigned32 *status);
```

ARGUMENTS

Input

protseq Specifies a protocol sequence to register with the RPC run-time system. Annex B lists valid protocol sequence identifiers that may be used for this argument.

max_call_requests Specifies the number of concurrent remote procedure call requests that the server is guaranteed to accept from the transport. The RPC run-time system allocates sufficient network resources to handle this number of concurrent calls.

The RPC run-time system guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of **max_call_requests** and can vary for each protocol sequence.

The value rpc_c_protseq_max_reqs_default specifies an implementation-dependent default value ≥1.

Note: The **rpc_server_listen**( ) routine specifies a **max_calls_exec** argument that specifies the number of call threads the server will allocate to handle calls. Normally, the values of **max_calls_exec** and **max_call_requests** are the same. Servers are guaranteed to support the minimum of **max_calls_exec** and **max_call_requests** concurrent remote procedure calls. Applications should not rely on a server handling more than this number.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

DESCRIPTION

The **rpc_server_use_protseq**( ) routine registers a single protocol sequence with the RPC run-time system. A server must register at least one protocol sequence with the RPC run-time system to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences.

The **max_call_requests** argument allows an application to specify the number of concurrent remote procedure call requests the server is guaranteed to handle.

Note: To register all stub-defined protocol sequences, a server calls the **rpc_server_use_all_protseqs**( ) routine.

For a list of routines typically called after **rpc_server_use_protseq**( ), see **rpc_server_use_all_protseqs**( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RETURN VALUE

None.

SEE ALSO

rpc_binding_vector_free()
rpc_ep_register()
rpc_ep_register_no_replace()
rpc_network_is_protseq_valid()
rpc_ns_binding_export()
rpc_server_inq_bindings()
rpc_server_listen()
rpc_server_register_if()
rpc_server_use_all_protseqs()
rpc_server_use_all_protseqs()
rpc_server_use_protseq_ep()
rpc_server_use_protseq_if().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_server_use_protseq_ep — tells the RPC run-time system to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls (used by server applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_use_protseq_ep(
      unsigned_char_t *protseq,
      unsigned32 max_call_requests,
      unsigned_char_t *endpoint,
      unsigned32 *status);
```

ARGUMENTS

Input

protseq Specifies a protocol sequence to register with the RPC run-time system. Annex B lists valid protocol sequence identifiers that may be used for this argument.

max_call_requests Specifies the number of concurrent remote procedure call requests that the server is guaranteed to accept from the transport. The RPC run-time system allocates sufficient network resources to handle this number of concurrent calls.

The RPC run-time system guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of **max_call_requests** and can vary for each protocol sequence.

The value rpc_c_protseq_max_reqs_default specifies an implementation-dependent default value ≥1.

Note: The **rpc_server_listen**() routine specifies a **max_calls_exec** argument that specifies the number of call threads the server will allocate to handle calls. Normally, the values of **max_calls_exec** and **max_call_requests** are the same. Servers are guaranteed to support the minimum of **max_calls_exec** and **max_call_requests** concurrent remote procedure calls. Applications should not rely on a server handling more than this number.

endpoint Specifies address information for an endpoint. This information is used in creating a binding handle for the protocol sequence specified in the **protseq** argument. (See Subclause 3.1 for information on the syntax of the **endpoint** argument.)

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

DESCRIPTION

The **rpc_server_use_protseq_ep**() routine registers a protocol sequence and its specified endpoint address information with the RPC run-time system. A server must register at least one protocol sequence with the RPC run-time system to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences and endpoints.

The **max_call_requests** argument specifies the number of concurrent remote procedure call

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

requests the server is guaranteed to handle.

Note: For a list of routines typically called after **rpc_server_use_protseq_ep**(), see **rpc_server_use_all_protseqs**().

RETURN VALUE

None.

SEE ALSO

rpc_binding_vector_free()
rpc_ep_register()
rpc_ep_register_no_replace()
rpc_ns_binding_export()
rpc_server_inq_bindings()
rpc_server_listen()
rpc_server_register_if()
rpc_server_use_all_protseqs()
rpc_server_use_all_protseqs()
rpc_server_use_protseq()
rpc_server_use_protseq_ep().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_server_use_protseq_if — tells the RPC run-time system to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls (used by server applications)

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_server_use_protseq_if(
        unsigned_char_t *protseq,
        unsigned32 max_call_requests,
        rpc_if_handle_t if_handle,
        unsigned32 *status);
```

ARGUMENTS

Input

protseq Specifies a protocol sequence to register with the RPC run-time system. Annex B lists valid protocol sequence identifiers that may be used for this argument.

max_call_requests Specifies the number of concurrent remote procedure call requests that the server is guaranteed to accept from the transport. The RPC run-time system allocates sufficient network resources to handle this number of concurrent calls.

The RPC run-time system guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of **max_call_requests** and can vary for each protocol sequence.

The value rpc_c_protseq_max_reqs_default specifies an implementation-dependent default value ≥1.

Note: The **rpc_server_listen** routine specifies a **max_calls_exec** argument that specifies the number of call threads the server will allocate to handle calls. Normally, the values of **max_calls_exec** and **max_call_requests** are the same. Servers are guaranteed to support the minimum of **max_calls_exec** and **max_call_requests** concurrent remote procedure calls. Applications should not rely on a server handling more than this number.

if_handle Specifies an interface specification whose endpoint information is used in creating a binding for the protocol sequence specified in the **protseq** argument.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

DESCRIPTION

The **rpc_server_use_protseq_if**() routine registers one protocol sequence with the RPC run-time system, including its endpoint address information as provided by the stub.

A server must register at least one protocol sequence with the RPC run-time system to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The **max_call_requests** argument specifies the number of concurrent remote procedure call requests the server is guaranteed to handle.

Note: To register all stub-specified protocol sequences, applications use the **rpc_server_use_all_protseqs**( ) routine. After calling **rpc_server_use_protseq_if**( ), an application typically calls the following routines:

- **rpc_server_inq_bindings**( ), which obtains a vector containing all of the server's binding handles
- **rpc_ns_binding_export**( ), which places the binding handles in the name service database for access by any client
- **rpc_binding_vector_free**( ), which frees the vector of server binding handles
- **rpc_server_register_if**( ), which registers with the RPC run-time system those interfaces that the server offers
- **rpc_server_listen**( ), which enables the reception of remote procedure calls.

RETURN VALUE

None.

SEE ALSO

rpc_binding_vector_free( )
rpc_ep_register( )
rpc_ep_register_no_replace( )
rpc_ns_binding_export( )
rpc_server_inq_bindings( )
rpc_server_listen( )
rpc_server_register_if( )
rpc_server_use_all_protseqs( )
rpc_server_use_all_protseqs( )
rpc_server_use_protseq( )
rpc_server_use_protseq_ep( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_allocate — allocates memory within the RPC stub memory management scheme

SYNOPSIS

```
#include <rpc.h>

idl_void_p_t rpc_sm_allocate(
        unsigned32 size,
        unsigned32 *status);
```

ARGUMENTS

Input

size Specifies, in bytes, the size of memory to be allocated.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

Applications call **rpc_sm_allocate**( ) to allocate memory within the RPC stub memory management scheme. Before a call to this routine, the stub memory management environment must have been established. For manager code that is called from the stub, the stub itself normally establishes the necessary environment. (See Section 2 for a description of stub memory management and an explanation of the conditions under which the stub establishes the necessary memory management environment.) When **rpc_sm_allocate**( ) is used by code that is not called from the stub, the application must establish the required memory management environment by calling **rpc_sm_enable_allocate**( ).

When the stub establishes the memory management environment, the stub itself frees any memory allocated by **rpc_sm_allocate**( ). The application can free such memory before returning to the calling stub by calling **rpc_sm_free**( ).

When the application establishes the memory management environment, it must free any memory allocated, either by calling **rpc_sm_free**( ) or by calling **rpc_sm_disable_allocate**.

Multiple threads may call **rpc_sm_allocate**( ) and **rpc_sm_free**( ) to manage the same memory within the stub memory management environment. To do so, the threads must share the same stub memory management thread handle. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle**( ) and **rpc_sm_set_thread_handle**( ).

RETURN VALUE

A pointer to the allocated memory.

SEE ALSO

rpc_sm_free( )
rpc_sm_enable_allocate( )
rpc_sm_disable_allocate( )
rpc_sm_get_thread_handle( )
rpc_sm_set_thread_handle( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_client_free — frees memory returned from a client stub

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_client_free(
        idl_void_p_t node_to_free,
        unsigned32 *status);
```

ARGUMENTS

Input

node_to_free Specifies a pointer to memory returned from a client stub.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_sm_client_free**() routine releases memory allocated and returned from a client stub. The thread calling **rpc_sm_client_free**() must have the same thread handle as the thread that made the RPC call. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle**() and **rpc_sm_set_thread_handle**().

This routine enables a routine to deallocate dynamically allocated memory returned by an RPC call without knowledge of the memory management environment from which it was called.

RETURN VALUE

None.

SEE ALSO

rpc_sm_free()
rpc_sm_get_thread_handle()
rpc_sm_set_client_alloc_free()
rpc_sm_set_thread_handle()
rpc_sm_swap_client_alloc_free().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_destroy_client_context — reclaims the client memory resources for a context handle, and makes the context handle null

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_destroy_client_context(
        idl_void_p_t p_unusable_context_handle,
        unsigned32 *status);
```

ARGUMENTS

Input

p_unusable_context_handle

Specifies the context handle that can no longer be accessed.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_sm_destroy_client_context**() routine is used by client applications to reclaim the client resources used in maintaining an active context handle. Applications call this routine after a communications error makes the context handle unusable.

When the **rpc_sm_destroy_client_context**() routine reclaims the memory resources, it also makes the context handle null.

RETURN VALUE

None.

SEE ALSO

rpc_sm_allocate()
rpc_sm_enable_allocate().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_enable_allocate — enables the stub memory management environment

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_enable_allocate(
        unsigned32 *status);
```

ARGUMENTS

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

Applications can call **rpc_sm_enable_allocate**( ) to establish a stub memory management environment in cases where one is not established by the stub itself. A stub memory management environment must be established before any calls are made to **rpc_sm_allocate**( ). For server manager code called from the stub, the stub memory management environment is normally established by the stub itself. (See Section 2 for a description of stub memory management and an explanation of the conditions under which the stub establishes the necessary memory management environment.) Code that is called from other contexts needs to call **rpc_sm_enable_allocate**( ) before calling **rpc_sm_allocate**( ).

Note: For a discussion of how spawned threads acquire a stub memory management environment, see **rpc_sm_get_thread_handle**( ).

RETURN VALUE

None.

SEE ALSO

rpc_sm_allocate( )
rpc_sm_disable_allocate( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_free — frees memory allocated by the **rpc_sm_allocate**( ) routine

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_free(
        idl_void_p_t node_to_free,
        unsigned32 *status);
```

ARGUMENTS

Input

node_to_free Specifies a pointer to memory allocated by **rpc_sm_allocate**( ).

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

Applications call **rpc_sm_free**( ) to release memory allocated by **rpc_sm_allocate**( ).

When the stub allocates memory within the stub memory management environment, manager code called from the stub can also use **rpc_sm_free**( ) to release memory allocated by the stub. (See Section 2 for a description of stub memory management.)

The thread calling **rpc_sm_free**( ) must have the same thread handle as the thread that allocated the memory with **rpc_sm_allocate**( ). Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle**( ) and **rpc_sm_set_thread_handle**( ).

RETURN VALUE

None.

SEE ALSO

rpc_sm_allocate( )
rpc_sm_get_thread_handle( )
rpc_sm_set_thread_handle( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_get_thread_handle — gets a thread handle for the stub memory management environment

SYNOPSIS

```
#include <rpc.h>

rpc_sm_thread_handle_t rpc_sm_get_thread_handle(
        unsigned32 *status);
```

ARGUMENTS

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

Applications call **rpc_sm_get_thread_handle**( ) to get a thread handle for the current stub memory management environment. A thread that is managing memory within the stub memory management scheme calls **rpc_sm_get_thread_handle**( ) to get a thread handle for its current stub memory management environment. A thread that calls **rpc_sm_set_thread_handle**( ) with this handle is able to use the same memory management environment.

When multiple threads call **rpc_sm_allocate**( ) and **rpc_sm_free**( ) to manage the same memory, they must share the same thread handle. The thread that established the stub memory management environment calls **rpc_sm_get_thread_handle**( ) to get a thread handle before spawning new threads that will manage the same memory. The spawned threads then call **rpc_sm_set_thread_handle**( ) with the handle provided by the parent thread.

Note: Typically, **rpc_sm_get_thread_handle** is called by a server manager routine before it spawns additional threads. Normally the stub sets up the memory management environment for the manager routine. The manager calls **rpc_sm_get_thread_handle** to make this environment available to the spawned threads.

A thread may also use **rpc_sm_get_thread_handle** and **rpc_sm_set_thread_handle** to save and restore its memory management environment.

RETURN VALUE

A thread handle.

SEE ALSO

rpc_sm_allocate( )
rpc_sm_free( )
rpc_sm_set_thread_handle( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_set_client_alloc_free — sets the memory allocation and freeing mechanisms used by the client stubs

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_set_client_alloc_free(
        idl_void_p_t (*p_allocate)(
            unsigned32 size),
        void (*p_free)(
            idl_void_p_t ptr),
        unsigned32 *status);
```

ARGUMENTS

Input

p_allocate Specifies a memory allocator routine.

p_free Specifies a memory free routine. This routine is used to free memory allocated with the routine specified by **p_allocate**.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_sm_set_client_alloc_free**( ) routine overrides the default routines that the client stub uses to manage memory.

Note: The default memory management routines are ISO **free**( ) and ISO **malloc**( ), except when the remote call occurs within manager code in which case the default memory management routines are **rpc_sm_free**( ) and **rpc_sm_allocate**( ).

RETURN VALUE

None.

SEE ALSO

rpc_sm_allocate( )
rpc_sm_free( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_set_thread_handle — sets a thread handle for the stub memory management environment

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_set_thread_handle(
        rpc_sm_thread_handle_t id,
        unsigned32 *status);
```

ARGUMENTS

Input

id Specifies a thread handle returned by a call to the **rpc_sm_get_thread_handle**( ) routine.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

An application thread calls **rpc_sm_set_thread_handle**( ) to set a thread handle for memory management within the stub memory management environment. A thread that is managing memory within the stub memory management scheme calls **rpc_sm_get_thread_handle**( ) to get a thread handle for its current stub memory management environment. A thread that calls **rpc_sm_set_thread_handle**( ) with this handle is able to use the same memory management environment.

When multiple threads call **rpc_sm_allocate**( ) and **rpc_sm_free**( ) to manage the same memory, they must share the same thread handle. The thread that established the stub memory management environment calls **rpc_sm_get_thread_handle**( ) to get a thread handle before spawning new threads that will manage the same memory. The spawned threads then call **rpc_sm_set_thread_handle**( ) with the handle provided by the parent thread.

Note: Typically, **rpc_sm_set_thread_handle**( ) is called by a thread spawned by a server manager routine. Normally, the stub sets up the memory management environment for the manager routine and the manager calls **rpc_sm_get_thread_handle**( ) to get a thread handle. Each spawned thread then calls **rpc_sm_get_thread_handle**( ) to get access to the manager's memory management environment.

A thread may also use **rpc_sm_get_thread_handle**( ) and **rpc_sm_set_thread_handle**( ) to save and restore its memory management environment.

RETURN VALUE

None.

SEE ALSO

rpc_sm_get_thread_handle( )
rpc_sm_allocate( )
rpc_sm_free( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_sm_swap_client_alloc_free — exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client

SYNOPSIS

```
#include <rpc.h>

void rpc_sm_swap_client_alloc_free(
        idl_void_p_t (*p_allocate)(
            unsigned32 size),
        void (*p_free)(
            idl_void_p_t ptr),
        idl_void_p_t (**p_p_old_allocate)(
            unsigned32 size),
        void (**p_p_old_free)(
            idl_void_p_t ptr),
        unsigned32 *status);
```

ARGUMENTS

Input

p_allocate	Specifies a new memory allocation routine.
p_free	Specifies a new memory free routine.

Output

p_p_old_allocate	Returns the memory allocation routine in use before the call to this routine.
p_p_old_free	Returns the memory free routine in use before the call to this routine.
status	Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not. Possible status codes and their meanings include: rpc_s_ok Success.

DESCRIPTION

The **rpc_sm_swap_client_alloc_free**() routine exchanges the current allocate and free mechanisms used by the client stubs for routines supplied by the caller.

RETURN VALUE

None.

SEE ALSO

rpc_sm_allocate()
rpc_sm_free()
rpc_sm_set_client_alloc_free().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_string_binding_compose — combines the components of a string binding into a string binding

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_string_binding_compose(
        unsigned_char_t *obj_uuid,
        unsigned_char_t *protseq,
        unsigned_char_t *network_addr,
        unsigned_char_t *endpoint,
        unsigned_char_t *options,
        unsigned_char_t **string_binding,
        unsigned32 *status);
```

ARGUMENTS

Input

obj_uuid	Specifies a string representation of an object UUID.
protseq	Specifies a representation of a protocol sequence.
network_addr	Specifies a string representation of a network address.
endpoint	Specifies a string representation of an endpoint.
options	Specifies a string representation of network options.

Input/Output

string_binding	Returns a pointer to a string representation of a binding. Specifying NULL prevents the routine from returning this argument. In this case, no string is allocated.

Output

status	Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not. Possible status codes and their meanings include: rpc_s_ok Success.

DESCRIPTION

The **rpc_string_binding_compose**( ) routine combines string binding handle components into a string binding. (See Subclause 3.1 for information on the syntax of string bindings.)

The RPC run-time system allocates memory for the string returned in **string_binding**. The application calls the **rpc_string_free**( ) routine to deallocate that memory.

A NULL or empty string ("") argument specifies that an input string that has no data.

RETURN VALUE

None.

SEE ALSO

rpc_binding_from_string_binding( )
rpc_binding_to_string_binding( )
rpc_string_binding_parse( )
rpc_string_free( )
uuid_to_string( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

rpc_string_binding_parse — returns, as separate strings, the object UUID and address components of a string binding

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_string_binding_parse(
        unsigned_char_t *string_binding,
        unsigned_char_t **obj_uuid,
        unsigned_char_t **protseq,
        unsigned_char_t **network_addr,
        unsigned_char_t **endpoint,
        unsigned_char_t **network_options,
        unsigned32 *status);
```

ARGUMENTS

Input

string_binding Specifies a string representation of a binding.

Input/Output

obj_uuid Returns a string representation of an object UUID.

Specifying NULL prevents the routine from returning this argument.

protseq Returns a string representation of a protocol sequence.

Specifying NULL prevents the routine from returning this argument.

network_addr Returns a string representation of a network address.

Specifying NULL prevents the routine from returning this argument.

endpoint Returns a string representation of an endpoint.

Specifying NULL prevents the routine from returning this argument.

network_options Returns a string representation of network options.

Specifying NULL prevents the routine from returning this argument.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_string_binding_parse**() routine parses a string representation of a binding into its component fields.

The RPC run-time system allocates memory for each component string the routine returns. To deallocate this memory, the application calls the **rpc_string_free**() routine once for each returned string. When NULL is specified for a component, no memory is allocated and no string is returned so the application need not call **rpc_string_free**().

When a field of the **string_binding** is empty, the **rpc_string_binding_parse**() routine returns the empty string ("") in the corresponding output.

RETURN VALUE

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

SEE ALSO

rpc_binding_from_string_binding()
rpc_binding_to_string_binding()
rpc_string_binding_compose()
rpc_string_free()
uuid_from_string().

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

rpc_string_free — frees a character string allocated by the run-time system

SYNOPSIS

```
#include <dce/rpc.h>

void rpc_string_free(
        unsigned_char_t **string,
        unsigned32 *status);
```

ARGUMENTS

Input/Output

string Specifies the address of the pointer to the character string to free.

The value NULL is returned.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

rpc_s_ok Success.

DESCRIPTION

The **rpc_string_free**() routine deallocates the memory occupied by a character string when it was allocated with one of the following routines:

dce_error_inq_text()
rpc_binding_inq_auth_client()
rpc_binding_inq_auth_info()
rpc_binding_to_string_binding()
rpc_mgmt_ep_elt_inq_next()
rpc_mgmt_inq_server_princ_name()
rpc_ns_binding_inq_entry_name()
rpc_ns_entry_expand_name()
rpc_ns_group_mbr_inq_next()
rpc_ns_profile_elt_inq_next()
rpc_string_binding_compose()
rpc_string_binding_parse()
uuid_to_string()

An application must call this routine once for each character string allocated and returned by calls to other RPC run-time routines.

RETURN VALUE

None.

SEE ALSO

dce_error_inq_text()
rpc_binding_inq_auth_client()
rpc_binding_inq_auth_info()
rpc_binding_to_string_binding()
rpc_mgmt_ep_elt_inq_next()
rpc_mgmt_inq_server_princ_name()
rpc_ns_binding_inq_entry_name()
rpc_ns_entry_expand_name()
rpc_ns_group_mbr_inq_next()
rpc_ns_profile_elt_inq_next()
rpc_string_binding_compose()
rpc_string_binding_parse()
uuid_to_string().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

uuid_compare — compares two UUIDs and determines their order

Used by client, server or management applications

SYNOPSIS

```
#include <dce/uuid.h>

signed32 uuid_compare(
        uuid_t *uuid1,
        uuid_t *uuid2,
        unsigned32 *status);
```

ARGUMENTS

Input

uuid1 A pointer to a UUID. This UUID is compared with the UUID specified in the **uuid2** argument.

uuid2 A pointer to a UUID. This UUID is compared with the UUID specified in the **uuid1** argument.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

uuid_s_ok Success.

DESCRIPTION

The **uuid_compare**() routine compares two UUIDs and determines their order.

RETURN VALUE

If successful, returns one of the following constants:

−1 The **uuid1** argument precedes the **uuid2** argument.

0 The **uuid1** argument is equal to the **uuid2** argument.

1 The **uuid1** argument follows the **uuid2** argument.

SEE ALSO

uuid_equal()
uuid_is_nil().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

uuid_create — creates a new UUID

SYNOPSIS

```
#include <dce/uuid.h>

void uuid_create(
        uuid_t *uuid,
        unsigned32 *status);
```

ARGUMENTS

Output

uuid	Returns the new UUID.
status	Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not. Possible status codes and their meanings include: uuid_s_ok Success.

DESCRIPTION

The **uuid_create**( ) routine creates a new UUID.

RETURN VALUE

None.

SEE ALSO

uuid_create_nil( )
uuid_from_string( )
uuid_to_string( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

uuid_create_nil — creates a nil-valued UUID

SYNOPSIS

```
#include <dce/uuid.h>

void uuid_create_nil(
        uuid_t *nil_uuid,
        unsigned32 *status);
```

ARGUMENTS

Output

nil_uuid	Returns a nil-valued UUID.
status	Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not. Possible status codes and their meanings include: uuid_s_ok Success.

DESCRIPTION

The **uuid_create_nil**( ) routine creates a nil-valued UUID.

RETURN VALUE

None.

SEE ALSO

uuid_create( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

uuid_equal — determines if two UUIDs are equal

SYNOPSIS

```
#include <dce/uuid.h>

boolean32 uuid_equal(
        uuid_t *uuid1,
        uuid_t *uuid2,
        unsigned32 *status);
```

ARGUMENTS

Input

uuid1 A pointer to a UUID. This UUID is compared with the UUID specified in the **uuid2** argument.

uuid2 A pointer to a UUID. This UUID is compared with the UUID specified in the **uuid1** argument.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

uuid_s_ok Success.

DESCRIPTION

The **uuid_equal**() routine compares two UUIDs and determines whether they are equal.

RETURN VALUE

If successful, returns one of the following constants:

TRUE The **uuid1** argument is equal to the **uuid2** argument.

FALSE The **uuid1** argument is not equal to the **uuid2** argument.

SEE ALSO

uuid_compare().

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

uuid_from_string — converts the string representation of a UUID to the binary representation

SYNOPSIS

```
#include <dce/uuid.h>

void uuid_from_string(
        unsigned_char_t *string_uuid,
        uuid_t *uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

string_uuid A string UUID. (For information about string UUIDs, see Subclause 3.1.)

Output

uuid Returns the UUID specified by the **string_uuid** argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

uuid_s_ok Success.

DESCRIPTION

An application calls the **uuid_from_string**( ) routine to convert the string representation of a UUID, **string_uuid**, to its equivalent binary representation.

RETURN VALUE

None.

SEE ALSO

uuid_to_string( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

uuid_is_nil — determines if a UUID is a nil-valued UUID.

SYNOPSIS

```
#include <dce/uuid.h>

boolean32 uuid_is_nil(
        uuid_t *uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

uuid Specifies a UUID to test for nil value.

Output

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

uuid_s_ok Success.

DESCRIPTION

The **uuid_is_nil**( ) routine determines whether the specified UUID is a nil-valued UUID. This routine yields the same result as if an application did the following:

- called the **uuid_create_nil**( ) routine
- called the **uuid_equal**( ) routine to compare the returned nil-value UUID to the UUID specified in the **uuid** argument.

RETURN VALUE

If successful, returns one of the following constants:

TRUE The **uuid** argument is a nil-valued UUID.

FALSE The **uuid** argument is not a nil-valued UUID.

SEE ALSO

uuid_compare( )
uuid_create_nil( )
uuid_equal( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NAME

uuid_to_string — converts a UUID from a binary representation to a string representation

SYNOPSIS

```
#include <dce/uuid.h>

void uuid_to_string(
        uuid_t *uuid,
        unsigned_char_t **string_uuid,
        unsigned32 *status);
```

ARGUMENTS

Input

uuid Specifies a UUID to be converted to string format.

Output

string_uuid Returns a pointer to the string representation of the UUID specified in the **uuid** argument.

status Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Possible status codes and their meanings include:

uuid_s_ok Success.

DESCRIPTION

The **uuid_to_string**( ) routine converts a UUID to string UUID.

Note: The RPC run-time system allocates memory for the string returned in **string_uuid**. To deallocate the memory, the application calls the **rpc_string_free**( ) routine.

RETURN VALUE

None.

SEE ALSO

rpc_string_free( )
uuid_from_string( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 4. Interface Definition Language

The Interface Definition Language (IDL) is a language for specifying operations (procedures or functions), parameters to these operations, and data types. This Section specifies IDL and the associated Attribute Configuration Source (ACS). It includes:

- a description of the notation used in the language specifications
- a specification of the IDL language
- a specification of the ACS
- a series of tables summarising IDL and ACS grammar.

4.1 Notation

The syntax of IDL and ACS is described using an extended BNF (Backus-Naur Form) notation. The meaning of the BNF notation is as follows:

- Brackets ([ ]) enclose an optional part of the syntax.
- Ellipsis points (...) indicate that the left clause can be repeated either zero or more times if it is optional or one or more times if it is required.
- The vertical bar (|) indicates alternative productions; it is read as "or".
- Language punctuation that does not conflict with punctuation characters used in the BNF notation appears in a production in the appropriate position. Language punctuation that does conflict with punctuation characters used in the BNF notation is enclosed in less-than and greater-than symbols; for example, <[>. Note particularly that when ' (single quotation) or " (double quotation) appear in a production, they are a part of the language and must appear in IDL source.

Elements in the grammar that are capitalised are terminals of the grammar. For example, <Identifier> is not further expanded. Also, keywords of the language are terminals of the grammar. For example, the keyword **boolean** is not further expanded.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2 IDL Language Specification

The syntax of the IDL language is derived from that of the ISO C programming language. Where a term in this description of the IDL language is not fully defined, the C-language definition of that term is implied.

This Section specifies both language syntax and semantics. As a result, the syntax is presented in pieces. Subclause 4.4.1 on page 247 provides a syntax summary, with the productions in the order in which they appear in this Section. An alphabetical cross-reference to the language syntax is also provided in Subclause 4.4.2 on page 250.

4.2.1 IDL Lexemes

The following subsubclauses define the lexemes of the IDL language.

4.2.1.1 Keywords and Reserved Words

The IDL contains keywords, which are listed in Subclause 4.6 on page 254. Some keywords are reserved words, and must not be used as identifiers. Keywords that are not reserved may be used as identifiers, except when used as attributes (that is, within **[ ]** (brackets)).

4.2.1.2 Identifiers

Each object is named with a unique identifier. The maximum length of an identifier is 31 characters.

Some identifiers are used as a base from which the compiler constructs other identifiers. These identifiers have further restrictions on their length. Table 4-3 on page 253 lists the classes and maximum lengths of identifiers that are used as bases. The character set for identifiers is the alphabetic characters A to Z and a to z, the digits 0 to 9, and the _ (underbar) character. An identifier must start with an alphabetic character or the _ (underbar) character.

The IDL is a case-sensitive language.

Restrictions on Names

Interface specifications must observe the following restrictions on names:

- A field name or parameter name cannot be the same as a type name.
- Type names, operation names, constant names and enumeration identifiers form a single name space. An identifier may only have one usage within this name space.

4.2.1.3 IDL Punctuation

The punctuation used in IDL consists of the following characters:

- the **.** (dot)
- the **,** (comma)
- the pair **( )** (parentheses)
- the pair **[ ]** (brackets)
- the pair **{ }** (braces)
- the **;** (semicolon)
- the **:** (colon)
- the ***** (asterisk)
- the **'** (single quote)
- the **"** (double quote)
- the **=** (equal sign).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.1.4 Alternate Representation of Braces

The **{** (open brace) and **}** (close brace) characters are defined as national replacement set characters by ISO and may not be present on all keyboards. Wherever an open brace is specified, the **??<** trigraph may be substituted. Wherever a close brace is specified, the **??>** trigraph may be substituted. These substitutions are the same as those specified in the ISO C standard.

4.2.1.5 White Space

White space is a character sequence that can be used to delimit any of the other low-level constructs. The syntax of white space is as follows:

- a blank
- a return
- a horizontal tab
- a form feed in column 1
- a comment
- a sequence of one or more white space constructs.

A language keyword, an `<Identifier>` or a list of `<Digit>`s must be preceded by a punctuation character or by white space. A language keyword, an `<Identifier>` or a list of `<Digit>`s must be followed by a punctuation character or by white space. Any punctuation character may be preceded or followed by white space.

4.2.2 Comments

The **/*** (slash and asterisk) characters introduce a comment. The contents of a comment are examined only to find the ***/** (asterisk and slash) that terminate it. Thus, comments do not nest. One or more comments may occur before the first non-comment lexical element of the IDL source, between any two lexical elements of the IDL source, or after the last non-comment lexical element of the IDL source.

4.2.3 Interface Definition Structure

An interface definition written in IDL has the following structure:

```
<interface> ::= <interface_header> { <interface_body> }
```

4.2.4 Interface Header

The structure of the interface header is as follows:

```
<interface_header> ::= <[> <interface_attributes> <]> interface <Identifier>
```

where:

```
<interface_attributes> ::= <interface_attribute>
       [ , <interface_attribute> ] ...
<interface_attribute> ::= uuid ( <Uuid_rep> )
       | version ( <Integer_literal>[.<Integer_literal>])
       | endpoint ( <port_spec> [ ,<port_spec> ] ... )
       | local
       | pointer_default ( <ptr_attr> )
<port_spec> ::=  <Family_string> :  <[> <Port_string> <]>
```

If an interface defines any operations, exactly one of the **uuid** attribute or the **local** attribute must be specified. Whichever is specified must appear exactly once.

It is permissible to have neither the **uuid** nor the **local** attribute if the interface defines no operations. The **version** attribute may occur at most once.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.4.1 The uuid Attribute

The **uuid** attribute designates the UUID that is assigned to the interface to identify it uniquely among all interfaces. The **uuid** attribute is expressed by the **uuid** keyword followed by a string of characters that gives the literal representation of the UUID.

The textual representation of a UUID is a string that consists of 8 hexadecimal digits followed by a dash, followed by 3 groups of 4 hexadecimal digits where the groups are separated by dashes, followed by a dash followed by 12 hexadecimal digits. Hexadecimal digits with alphabetic representations may use upper case or lower case. The following is an example of the textual representation of a UUID:

```
12345678-9012-b456-8001-080020b033d7
```

All UUIDs are machine generated in a manner that gives high guarantees of uniqueness. For an architectural definition of UUIDs and a full discussion of their guarantees, refer to Annex A.

4.2.4.2 The version Attribute

The **version** attribute identifies a specific version of a remote interface when multiple versions of the interface exist. Version semantics are specified in Section 6. The **version** attribute is represented by the **version** keyword followed by a decimal integer that represents the major version number of the interface, or two decimal integers separated by a dot, where the first integer represents the major version number and the second represents the minor version number. White space is not allowed between the two integers and the dot.

Legal Values for Version Numbers

The major and minor version numbers of an interface must each have a value between 0 and 65,535, inclusive.

Version Number Defaults

The defaulting rules for interface version numbers are as follows:

- The interpretation of an interface with only a single (major) version number is that the minor version number is 0.
- An interface with no **version** attribute is given the version number 0.0.

4.2.4.3 The endpoint Attribute

The **endpoint** attribute specifies the well-known endpoint(s) on which servers that export the interface will listen. Well-known endpoint values are typically assigned by the central authority that owns the protocol.

A `<port_spec>` is composed of two strings separated by punctuation as described in the syntax. The `<Family_string>` identifies the protocol family to be used; the `<Port_string>` identifies the well-known endpoint designation for that family.

The actual syntax of `<Port_string>` is family-dependent. Registered values of `<Port_string>` are given in Annex H.

4.2.4.4 The local Attribute

The **local** attribute provides a means to use the IDL compiler as a header generation language. When the local attribute is specified, the **uuid** attribute must not be specified.

Stubs are not generated for local interfaces. Checks for data transmissibility are omitted.

4.2.4.5 The pointer_default Attribute

The **pointer_default** attribute specifies the default treatment for pointers. If no **pointer_default** attribute is specified in the interface attributes, and a construct requiring a default pointer class is used, the compiler will issue an error. (See Subclause 4.2.20 on page 234 for more information.)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.5 Interface Body

The structure of the interface body is as follows:

```
<interface_body> ::= [ <import> ... ] <interface_component>
       [ <interface_component> ... ]
```

where:

```
<import> ::= import <import_list> ;
<interface_component> ::= <export>
       | <op_declarator> ;
<export> ::= <type_declarator> ;
       | <const_declarator> ;
       | <tagged_declarator> ;
```

4.2.6 Import Declaration

The syntax of the <import_list> in an import declaration is as follows:

```
<import_list> ::= <import_name> [ , <import_name> ] ...
<import_name> ::=  "<Import_string>"
```

where: <Import_string> gives a system dependent name for an import source.

4.2.7 Constant Declaration

The following subsubclauses specify the syntax and semantics of constant declarations.

4.2.7.1 Syntax

The syntax for a constant declaration is as follows:

```
<const_declarator> ::= const <const_type_spec> <Identifier> = <const_exp>

<const_type_spec> ::= <primitive_integer_type>
       | char
       | boolean
       | void *
       | char *
<const_exp> ::= <integer_const_exp>
       | <Identifier>
       | <string>
       | <character_constant>
       | NULL
       | TRUE
       | FALSE
<integer_const_exp> ::= <conditional_exp>
<conditional_exp> ::= <logical_or_exp>
       | <logical_or_exp> ? <integer_const_exp> : <conditional_exp>
<logical_or_exp> ::= <logical_and_exp>
       | <logical_or_exp> <||> <logical_and_exp>
<logical_and_exp> ::= <inclusive_or_exp>
       | <logical_and_exp> && <inclusive_or_exp>
<inclusive_or_exp> ::= <exclusive_or_exp>
       | <inclusive_or_exp> <|> <exclusive_or_exp>
<exclusive_or_exp> ::= <and_exp>
       | <exclusive_or_exp> ^ <and_exp>
<and_exp> ::= <equality_exp>
       | <and_exp> & <equality_exp>
<equality_exp> ::= <relational_exp>
       | <equality_exp> == <relational_exp>
       | <equality_exp> != <relational_exp>
<relational_exp> ::= <shift_exp>
       | <relational_exp> <<> <shift_exp>
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        | <relational_exp> <>> <shift_exp>
        | <relational_exp> <<=> <shift_exp>
        | <relational_exp> <>=> <shift_exp>
<shift_exp> ::= <additive_exp>
        | <shift_exp> <<<> <additive_exp>
        | <shift_exp> <>>> <additive_exp>

<additive_exp> ::= <multiplicative_exp>
        | <additive_exp> + <multiplicative_exp>
        | <additive_exp> - <multiplicative_exp>
<multiplicative_exp> ::= <unary_exp>
        | <multiplicative_exp> * <unary_exp>
        | <multiplicative_exp> / <unary_exp>
        | <multiplicative_exp> % <unary_exp>
<unary_exp> ::= <primary_exp>
        | + <primary_exp>
        | - <primary_exp>
        | ~ <primary_exp>
        | ! <primary_exp>
<primary_exp> ::= <Integer_literal>
        | <Identifier>
        | '(' <const_exp> ')'
<string> ::= "[<Character>] ... "
<character_constant> ::= '<Character>'
```

4.2.7.2 Semantics and Restrictions

The <integer_size> keyword **hyper** must not appear in a constant declaration. <Character> is a character from the portable character set specified in Annex G.

In the production <string>, no <Character> is permitted to be the " (double quote) character, which is the delimiter, unless it is immediately preceded by the \ (backslash) character. In the production <character_constant>, no <Character> may be the ' (single quote) character, which is the delimiter, unless it is immediately preceded by the \ (backslash) character.

<Integer_literal> may appear if and only if <const_type_spec> is **long**, **short**, **small**, **unsigned long**, **unsigned short** or **unsigned small**.

A <character_constant> may appear if and only if <const_type_spec> is **char**. NULL may appear if and only if <const_type_spec> is **void***.

TRUE or FALSE may appear if and only if <const_type_spec> is **boolean**.

<string> may appear if and only if <const_type_spec> is **char***. Within a <string> a \ (backslash) is interpreted as introducing an escape sequence, as defined in ISO C standard, Subclause 3.1.3.4. White space within a <string> is significant and is preserved as part of the text of the string.

An <Identifier> must have been defined in a preceding constant declaration. The type that <Identifier> was defined to be in that declaration must be consistent with the <const_type_spec> in the current declaration.

4.2.8 Type Declarations and Tagged Declarations

The syntax for type declarations and tagged declarations is as follows:

```
<type_declarator> ::= typedef [ <type_attribute_list> ] <type_spec>
      <declarators>
<type_attribute_list> ::= <[>  <type_attribute>
      [ , <type_attribute> ] ... <]>
<type_spec> ::= <simple_type_spec>
      | <constructed_type_spec>
<simple_type_spec> ::= <base_type_spec>
      | <predefined_type_spec>
      | <Identifier>
<declarators> ::= <declarator> [ , <declarator> ] ...
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
<declarator> ::= <pointer_opt> <direct_declarator>
<direct_declarator> ::= <Identifier>
       | ( <declarator> )
       | <array_declarator>
       | <function_declarator>
<tagged_declarator> ::= <tagged_struct_declarator>
       | <tagged_union_declarator>
```

If a <simple_type_spec> is an <Identifier>, that <Identifier> must have been defined previously.

4.2.9 Base Types

The base types are the fundamental data types of the IDL. Any other data types in an interface definition are derived from these types. Subclause 4.2.9.1 gives the syntax rules for the base types. Subclause 4.2.9.2 to Subclause 4.2.9.7 on page 226 define the various types.

4.2.9.1 Syntax

The syntax rules for use of the base types are as follows:

```
<base_type_spec> ::= <floating_pt_type>
       | <integer_type>
       | <char_type>
       | <boolean_type>
       | <byte_type>
       | <void_type>
       | <handle_type>
<floating_pt_type> ::= float
       | double
<integer_type> ::= <primitive_integer_type>
       | hyper [unsigned] [int]
       | unsigned hyper [int]
<primitive_integer_type> ::= <signed_integer>
       | <unsigned_integer>
<signed_integer> ::= <integer_size> [ int ]
<unsigned_integer> ::= <integer_size> unsigned [ int ]
       | unsigned <integer_size> [ int ]
<integer_size> ::=  long
       | short
       | small
<char_type> ::= [ unsigned ] char
<boolean_type> ::= boolean
<byte_type> ::= byte
<void_type> ::= void
<handle_type> ::= handle_t
```

4.2.9.2 Integer Types

Table 4-1 on page 226 lists the integer types and their ranges.

Table 4-1 Integer Base Types

Type	Range
hyper	$-2^{63} \le 2^{63}-1$
long	$-2^{31} \le 2^{31}-1$
short	$-2^{15} \le 2^{15}-1$
small	$-2^{7} \le 2^{7}-1$
unsigned hyper	$0 \le 2^{64}-1$
unsigned long	$0 \le 2^{32}-1$
unsigned short	$0 \le 2^{16}-1$
unsigned small	$0 \le 2^{8}-1$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.2.9.3 The char Types

The keywords **char** and **unsigned char** are synonymous. Annex G contains portable character set values.

4.2.9.4 The boolean Type

The **boolean** keyword is used to indicate a data item that can only take one of the values TRUE and FALSE.

4.2.9.5 The byte Type

A **byte** data item consists of opaque data: that is, its contents are not interpreted.

4.2.9.6 The void Type

The **void** keyword is valid only in an operation or pointer declaration. In an operation declaration, it may be used to indicate an operation that does not return a function result value. In a pointer declaration, it must be used in conjunction with the **context_handle** attribute.

4.2.9.7 The handle_t Type

The **handle_t** type is used to declare a primitive handle object. A primitive handle can be declared in a type declaration or in a parameter list. If it is declared in a parameter list, it must be the first parameter in the list.

4.2.10 Constructed Types

The syntax of constructed types is as follows:

```
<constructed_type_spec> ::= <struct_type>
        | <union_type>
        | <enumeration_type>
        | <tagged_declarator>
        | <pipe_type>
```

4.2.11 Structures

The syntax of structures is as follows:

```
<tagged_struct_declarator> ::= struct <tag>
        | <tagged_struct>
<struct_type> ::= struct  { <member_list> }
<tagged_struct> ::= struct <tag> { <member_list> }
<tag> ::= <Identifier>
<member_list> ::= <member> [ <member> ] ...
<member> ::= <field_declarator> ;
<field_declarator> ::= [ <field_attribute_list> ] <type_spec> <declarators>
<field_attribute_list> ::= <[> <field_attribute>
        [ , <field_attribute> ] ... <]>
```

ISO C semantics apply to IDL structures.

4.2.12 Unions

The following subsubclauses describe IDL unions.

4.2.12.1 Syntax

The syntax for a union definition is:

```
<tagged_union_declarator> ::= union <tag>
        | <tagged_union>

<union_type> ::= union <union_switch> { <union_body> }
        | union { <union_body_n_e> }
<union_switch> ::= switch ( <switch_type_spec> <Identifier> )
        [ <union_name> ]
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
<switch_type_spec> ::= <primitive_integer_type>
       | <char_type>
       | <boolean_type>
       | <Identifier>
<tagged_union> ::= union <tag> <union_switch> { <union_body> }
       | union <tag> { <union_body_n_e> }
<union_name> ::= <Identifier>
<union_body> ::= <union_case> [  <union_case> ] ...
<union_body_n_e> ::=  <union_case_n_e> [ <union_case_n_e> ] ...
<union_case> ::= <union_case_label> [ <union_case_label> ] ... <union_arm>
       | <default_case>
<union_case_n_e> ::= <union_case_label_n_e> <union_arm>
                           | <default_case_n_e>
<union_case_label> ::= case <const_exp> :
<union_case_label_n_e> ::= <[> case ( <const_exp>
       [ , <const_exp> ] ... ) <]>
<default_case> ::= default : <union_arm>
<default_case_n_e> ::= <[> default <]> <union_arm>
<union_arm> ::= [ <field_declarator> ] ;
<union_type_switch_attr> ::= switch_type (  <switch_type_spec> )
<union_instance_switch_attr> ::= switch_is ( <attr_var> )
```

Encapsulated Unions

Encapsulated unions are created with the <union_switch> production.

Note: Encapsulated unions are so named because the discriminant and the union are tightly bound; that is, in a typical implementation they are both automatically encapsulated in a single structure.

Non-encapsulated Unions

A union that is created without the use of the <union_switch> production is a non-encapsulated union. The discriminant of a non-encapsulated union is another parameter if the union is a parameter, or another structure field if the union is a structure field.

When the non-encapsulated union is declared as a type, the <union_type_switch_attr> production must be used. When a type that is a non-encapsulated type is used to declare a structure field or a parameter, the <union_instance_switch_attr> production must be used. When a non-encapsulated union is being declared directly as a structure field or parameter, the <union_instance_switch_attr> production must be used.

4.2.12.2 Semantics and Restrictions

In encapsulated unions, if the <union_name> is omitted, the union is assigned the name **tagged_union** in the generated header source.

The <default_case> defines the layout of data if the discriminant variable of the **switch** is not equal to any of the **case** values.

Within a union, the type of each <union_case_label> must be that specified by the <switch_type_spec>. Likewise the type specified in the <union_type_switch_attr> and the <union_instance_switch_attr> must be the same.

A field within a union definition must not be or contain a conformant or conformant varying array. (See Subclause 4.2.15 on page 228 for descriptions of conformant and conformant varying arrays.)

A union arm that consists solely of a terminating semicolon is legal and specifies a null arm.

There must be at most one default case for a union.

When the <Identifier> alternative is used in the <switch_type_spec> production to declare a switch type, <Identifier> must only identify a declared type that resolves to on of the primitive integer type, the char type, the boolean type, or an enum type; it cannot be a pointer, struct, or array type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.13 Enumerated Types

The syntax of enumerated types is as follows:

```
<enumeration_type> ::= enum { <Identifier> [ , <Identifier> ] ... }
```

The <Identifier>s are mapped from left to right onto consecutive integers, beginning with the value zero.

An enumeration may have up to 32,767 <Identifier>s.

4.2.14 Pipes

IDL supports streams of typed data. The programming construct to support this is **pipe**, which is a type constructor that is similar to **struct** and **union**.

An **in** parameter that is a pipe allows a callee to pull an open-ended stream of typed data from a caller. An **out** parameter that is a pipe allows a callee to push an open-ended stream back to a caller.

4.2.14.1 Syntax

The syntax used to declare a pipe type is as follows:

```
<pipe_type> ::= pipe <type_spec>
```

4.2.14.2 Semantics and Restrictions

Data types that are pipes are subject to several restrictions:

- Pipe types must only be parameters; that is, a pipe type must not be the base type of an array or a pipe, a function result or a member of a structure or union.
- The base type of a pipe must not be or contain a pointer, a conformant array or a conformant structure.
- A pipe type must not be used in the definition of another type.
- A pipe type must not have the **transmit_as** attribute.
- A type that is the base type for a pipe must not have the **transmit_as** attribute.
- A pipe parameter must be passed by value or by reference. A pipe that is passed by reference must not have the **ptr** parameter attribute.
- A pipe type must not be the target of a pointer.

Also, an operation that has one or more pipe parameters must not have the **idempotent** attribute.

4.2.15 Arrays

The following subclauses describe the syntax and semantics of arrays.

4.2.15.1 Syntax

The syntax rules for array declarations are as follows:

```
<array_declarator> ::= <direct_declarator> <array_bounds_declarator>
<array_bounds_declarator> ::= <[> [ <array_bound> ] <]>
       | <[> <array_bounds_pair> <]>
<array_bounds_pair> ::= <array_bound> .. <array_bound>

<array_bound> ::= *
       | <integer_const_expr>
       | <Identifier>
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.15.2 Semantics and Restrictions

The bounds of each dimension of an array are expressed inside a separate < [>, <] > (bracket) pair.

If the bracket pair contains a single <const_exp> that evaluates to **n**, a lower bound of zero and an upper bound of **n**–1 are signified.

If the bracket pair is empty or contains a single * (asterisk), a lower bound of zero and an upper bound to be determined at run time are signified.

In an <array_bounds_pair> the <array_bound> preceding the .. (dot dot) indicates the lower bound of the dimension and the <array_bound> following the .. (dot dot) indicates the upper bound.

An * (asterisk) before or after .. (dot dot) means that the corresponding bound is to be determined at run time.

Any <Identifier> appearing as an array bound must resolve to an integer constant. The base type of an array cannot be a conformant array or conformant structure.

4.2.15.3 Arrays of Arrays

The user may declare types that are arrays and then declare arrays of objects of such types. The semantics of an **m**-dimensional array of objects of a type defined to be an **n**-dimensional array are the same as the semantics of an **m+n**-dimensional array.

4.2.16 Type Attributes

The following subclauses describe type attributes.

4.2.16.1 Syntax

The syntax for type attributes is as follows:

```
<type_attribute> ::= transmit_as ( <xmit_type> )
       | handle
       | align ( <integer_size> )
       | <usage_attribute>
       | <union_type_switch_attr>
       | <ptr_attr>
<usage_attribute> ::= string
       | context_handle
<xmit_type> ::= <simple_type_spec>
```

4.2.16.2 Semantics and Restrictions

Attributes that are specified for a type in a **typedef** statement are inherited by declarations that specify that type. A parameter of a call inherits the attributes of its corresponding type.

4.2.16.3 The transmit_as Attribute

The **transmit_as** attribute associates a presented type in the target language with an IDL transmitted type, <xmit_type>. The user must supply routines that perform conversions between the presented and transmitted types, and that release memory used to hold the converted data.

The <xmit_type> in a **transmit_as** attribute must be either a <base_type_spec>, a <predefined_type_spec> or an <Identifier> from a <type_declarator> that was previously defined.

The following types must not have the **transmit_as** attribute.

- types with the **context_handle** attribute; this also applies to types used as a parameter that has the **context_handle** attribute
- pipe types
- types used as the base type in a pipe definition

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- conformant, varying or conformant varying arrays
- conformant structures; the following restrictions apply to the transmitted type:
  - — If an **out** or **in, out** parameter is a type with the **transmit_as** attribute, the transmitted type must not be conformant.
  - — The transmitted type must not be a pointer or contain a pointer.

Subclause 5.1.5.3 on page 263 for the interaction of the **transmit_as** and **handle** attributes.

4.2.16.4 The handle Attribute

The **handle** attribute specifies that a type may serve as a customised handle. Customised handles permit the design of handles that are meaningful to an application. The user must provide binding and unbinding routines to convert between the custom handle type and the primitive handle type, **handle_t**.

4.2.16.5 The string Attribute

The **string** attribute is provided to preserve the string property of data so that it may be appropriately processed by application components written in languages that support a string data type.

The **string** attribute only applies to one-dimensional arrays. The element type of such an array is limited to one of the following:

a) **char**

b) **byte**

c) A structure all of whose members are of type **byte**, or a named type that resolves to **byte**. For this purpose, the NULL-terminated string construct supported in C through the **str**... library routines serves as a string data type. If the base type of a string is larger than one byte, then the required NULL terminator must be composed of the same number of bytes as the base type.

d) A named type that resolves to one of items 1, 2 or 3.

For further information on objects with the **string** attribute, see Subclause 4.2.19 on page 233.

4.2.16.6 The context_handle Attribute

In many interfaces, a called procedure needs to maintain state between calls. This is done by means of a **context handle**. A context handle is a **void*** with the **context_handle** attribute.

There are several restrictions on the use of context handles.

- Context handles must only be parameters. They must not be array elements, structure or union members, or the base type of a pipe.
- Context handles must not have the **transmit_as** attribute.
- Context handles must not have the **ptr** attribute.
- A parameter that is or contains a context handle must not have the **ptr** attribute.
- A NULL context handle must not be **in** only.
- A context handle must not be the target of a pointer.

4.2.17 Field Attributes

The following subclauses describe the syntax and semantics of field attributes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.17.1 Syntax

The syntax for declaring field attributes is as follows:

```
<field_attribute> ::= first_is ( <attr_var_list> )
        | last_is ( <attr_var_list> )
        | length_is ( <attr_var_list> )
        | min_is ( <attr_var_list> )
        | max_is ( <attr_var_list> )
        | size_is ( <attr_var_list> )
        | <usage_attribute>
        | <union_instance_switch_attr>
        | ignore
        | <ptr_attr>

<attr_var_list> ::= <attr_var> [ , <attr_var> ] ...
<attr_var> ::= [ [ * ] <Identifier> ]
```

4.2.17.2 Semantics and Restrictions

A <usage_attribute> must not be applied to an item of a type that resolves to a type with a <usage_attribute> in its declaration. For the form *<Identifier>, the <Identifier> in the <attr_var> must be a pointer to a <primitive_integer_type>. The **context_handle** attribute is not permitted on fields. Therefore, the <usage_attribute> must not resolve to **context_handle** in this production.

4.2.17.3 The ignore Attribute

The **ignore** attribute is used for a pointer, contained within a structure, that must not be dereferenced. The **ignore** attribute is restricted to pointer members of structures.

4.2.18 Field Attributes in Array Declarations

Field attributes are used in conjunction with array declarations to specify either the size of the array, or the portion of the array that contains valid data. To do this, they associate another parameter or structure field with the array. The associated datum contains the extra information. Array parameters must associate with another parameter, and array structure fields must associate with another field. Array parameters with field attributes are not allowed to have a **transmit_as** attribute associated with their type. Since an array parameter with one or more of the field attributes **max_is**, **size_is**, **first_is**, **last_is** or **length_is** is fully described only in the presence of another parameter, it cannot be properly processed by the <type_id>_to_xmit_type routine.

4.2.18.1 Conformant Arrays

An array is called conformant if it has an <array_bounds_declarator> that is empty or contains an * (asterisk). Conformant arrays have one of the attributes **max_is** or **size_is**.

The max_is Attribute

The **max_is** attribute is used to specify the maximum array indexes in each dimension of an array. Each <attr_var> in a **max_is** clause specifies the maximum array index in one dimension.

An <attr_var> must be non-NULL if and only if the upper bound of the corresponding dimension is empty or is an * (asterisk). At least one <attr_var> in a **max_is** clause must be non-NULL.

The <attr_var>s in a **max_is** clause are in one-to-one correspondence with the dimensions of the array, with the first <attr_var> corresponding to the first dimension. If there are fewer <attr_var>s than the array has dimensions, then the behaviour is undefined.

If a declaration has the **max_is** attribute, it must not have the **size_is** attribute. (See **The size_is Attribute** on page 232 for the relationship between these two attributes.)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The size_is Attribute

The **size_is** attribute is used to specify the number of data elements in each dimension of an array. Each <attr_var> in a **size_is** clause specifies the number of data elements in one dimension.

An <attr_var> must be non-NULL if and only if the upper bound of the corresponding dimension is empty or contains an * (asterisk). At least one <attr_var> in a **size_is** clause must be non-NULL.

The <attr_var>s are in one-to-one correspondence with the dimensions of the array, with the first <attr_var> corresponding to the first dimension. If there are fewer <attr_var>s than the array has dimensions, then the behaviour is undefined.

If a declaration has a **size_is** attribute, it must not have a **max_is** attribute. A given conformant array may be declared using either the **max_is** or **size_is** attribute. The relationship between the values of these attributes for a given array can be derived as follows: in the **m**th dimension of the array, the lowest index to be used in this dimension is specified by **lower_bound_m**. For a declaration using the **max_is** attribute, call the the **m**th <attr_var> **max_value_m**. For a declaration of the same array using the **size_is** attribute, call the **m**th <attr_var> **size_value_m**. For equivalent declarations, the relationship between these attribute values is then given by the equation:

```
size_value_m = max_value_m - lower_bound_m + 1
```

4.2.18.2 Varying and Conformant Varying Arrays

An array is called **varying** if none of its <array_bounds_declarator> components is empty or contains an * (asterisk), and it has either a **last_is**, **first_is** or **length_is** attribute. An array is called **conformant varying** if it is conformant and it has a **last_is**, **first_is** or **length_is** attribute.

The last_is Attribute

The **last_is** attribute is used to define the upper index bounds for significant elements in each dimension of an array. Each <attr_var> in a **last_is** clause indicates the highest index, in a given dimension, whose element value is significant both to the caller and the callee. Elements in a given dimension with indexes higher than the dimension's <attr_var> are not meaningful to caller and callee.

If the **<attr_var>** corresponding to a dimension in an array is null, then the value used is the corresponding dimension found in either the associated type definition for a varying array, or the value of the **max_is** or **size_is** parameter for a conformant varying array.

The <attr_var>s are in one-to-one correspondence with the dimensions of the array, with the first <attr_var> corresponding to the first dimension. If there are fewer <attr_var>s than the array has dimensions, then the behaviour is undefined.

If a declaration has a **last_is** attribute, it must not have a **length_is** attribute. (See **The length_is Attribute** on page 233 for the relationship between these two attributes.)

The first_is Attribute

The **first_is** attribute is used to define the lower index bounds for significant elements in each dimension of an array. Each <attr_var> in a **first_is** clause indicates the lowest index, in a given dimension, whose element value is significant both to the caller and the callee. Elements in a given dimension with indexes lower than the dimension's <attr_var> are not meaningful to caller and callee.

A NULL <attr_var> indicates that the lower bound for that dimension is to be used as the **first_is** value for that dimension. At least one <attr_var> in a **first_is** clause must be non-NULL.

The <attr_var>s are in one-to-one correspondence with the dimensions of the array, with the first <attr_var> corresponding to the first dimension. If there are fewer <attr_var>s than the array has dimensions, then the behaviour is undefined.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The length_is Attribute

The **length_is** attribute is used to define the number of significant elements in each dimension of an array. Each <attr_var> in a **length_is** clause indicates the number of elements, in a given dimension, whose element value is significant both to the caller and the callee.

Significant elements in a given dimension are counted from the lowest significant index for that dimension. This may be the fixed lower bound for this dimension or may be specified by using a **first_is** clause.

At least one <attr_var> in a **length_is** clause must be non-NULL.

If a declaration has a **length_is** attribute, it must not have a **last_is** attribute.

A given varying array may be declared using either the **last_is** or **length_is** attribute. The relationship between the values of these attributes for a given array is derived as follows: in the **m**th dimension of the array, the lowest index to be used in this dimension is specified by **lowest_index_m**. For a declaration using the **last_is** attribute, call the the **m**th <attr_var> **last_value_m**. For a declaration of the same array using the **length_is** attribute, call the **m**th <attr_var> **length_value_m**. For equivalent declarations, the relationship between these attribute values is then given by the equation:

```
length_value_m = last_value_m - lowest_index_m + 1
```

4.2.18.3 Relationships Between Attributes

The following rules apply to the relationship between the **max_is**, **first_is** and **last_is** values for a dimension.

- The **first_is** value must not be greater than the **max_is** value. Otherwise, the behaviour is undefined.
- The **last_is** value must not be greater than the **max_is** value. Otherwise, the behaviour is undefined.
- If the **first_is** value is equal to the **last_is** value +1, the interpretation is to select zero elements.
- If the **first_is** value is greater than the **last_is** value +1, the behaviour is undefined.

4.2.18.4 Negative Size and Length Specifications

If the **size_is** or **length_is** value is negative, the behaviour is undefined.

4.2.19 Field Attributes in String Declarations

The **string** attribute provides guidance to the stub generator that an array must be treated as a string when generating stubs for languages that support strings. When generating stubs for languages that do not support strings, they are treated simply as an array of the base type; that is, the **string** attribute is ignored.

When declaring a string, it is necessary to declare space for one more than the maximum number of characters the string is to hold.

The following subsubclauses describe the use of each of the field attributes with strings.

4.2.19.1 The first_is, last_is and length_is Attributes

An array with the **string** attribute must not have any of the varying attributes; that is, **first_is**, **last_is** or **length_is**.

4.2.19.2 The max_is Attribute

The **max_is** attribute names a parameter or record field that holds one more than the highest allowable index for a data character of a string.

A conformant array with the **string** attribute need not have a **max_is** or **size_is** attribute if and only if it is an **in** or **in,out** parameter. In these cases the extent is be determined from the input length.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.2.19.3 The size_is Attribute

A **size_is** attribute is an alternative mechanism for defining the maximum amount of string data that may be present. If a declaration has a **size_is** attribute, it must not have a **max_is** attribute.

The usage of **size_is** and its relationship to **max_is** must be derived from **The size_is Attribute** on page 232 by using the obvious analogy between arrays with the string property and those without.

4.2.20 Pointers

The following subclauses describe pointers in IDL.

4.2.20.1 Syntax

The syntax for a pointer declaration is as follows:

```
<declarator> ::= <pointer_opt> <direct_declarator>
<pointer_opt> ::= [<pointer>]
<pointer> ::= *...
```

Elsewhere in the grammar, attributes are applied to entities involving pointer declaration. The production for pointer attributes is

```
<ptr_attr> ::= ref
        | ptr
```

A <ptr_attr> may be applied only to declarations with an explicit pointer declaration, or to the implicit pointer present for array parameters.

4.2.20.2 Semantics and Restrictions

Pointers are used in applications in a wide variety of ways. IDL currently supports two different levels of pointer capability:

- reference pointers
- full pointers.

Description of Reference Pointers

A reference pointer is one that is used for simple indirection. It has the following characteristics in any language that supports pointers:

- A reference pointer must not have the value NULL. It can always be dereferenced.
- A reference pointer's value must not change during a call. It always points to the same referent on return from the call as it did when the call was made.
- A referent pointed to by a reference pointer must not be reached from any other name in the operation; that is, the pointer must not cause any aliasing of data within the operation.
- For **in** and **in, out** parameters, data returned from the callee is written into the existing referent specified by the reference pointer.

If these restrictions are not met, the effects are undefined.

Description of Full Pointers

A full pointer has a wider range of capabilities. The characteristics of a full pointer are as follows:

- A full pointer may have the value NULL.
- A full pointer's value may change across a call. It may change from NULL to non-NULL, from non-NULL to NULL, or from one non-NULL value to a different non-NULL value. The value may also remain unchanged across the call.
- The referent pointed to by a full pointer may be reached from other names in the application; that is, full pointers support aliasing and cycles.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.20.3 Attributes Applicable to Pointers

The following IDL attributes are used to indicate the supported pointer classes:

ref Representing reference pointers

ptr Representing full pointers

These attributes may be used on parameters, structure and union members, and in type definitions.

Pointer Attributes in the Interface Header

At most, one <ptr_attr> may appear in the interface header. (This appears with the **pointer_default** attribute. See Subclause 4.2.4.5 on page 222.) A <ptr_attr> must appear if any of the following is true:

- The interface declaration contains an operation parameter, typedef, structure member, union arm, and so on, with more than one * (asterisk) in a pointer declaration.
- The interface declaration contains a **typedef**, structure member or union arm with a pointer declaration that does not have a <ptr_attr>.

If a <ptr_attr> that is not required according to the preceding rules appears in the interface header, it is ignored.

Pointer Attributes on Parameters

By default, the first[2] indirection operator (an *, asterisk) in a parameter declaration is treated as a reference pointer. This is the pointer that accomplishes what is commonly termed "pass by reference". If there are multiple indirection operators in a parameter declaration, all but the first have the characteristic specified by <ptr_attr> in the interface header.

Any pointer attributes placed on a parameter (that is, directly in the syntax of an operation declaration) affect only the first pointer declaration. To affect any others, intermediate named types must be used.

Pointer Attributes on Function Results

If an operation returns a pointer to a type, then **ptr** is permitted as an operation attribute to describe the properties of the pointer. Even in the absence of this attribute, the meaning is that a full pointer is returned by the function. No pointer attribute other than **ptr** is permitted as an operation attribute.

Pointer Attributes in Member Declarations

By default, pointers in structure member and union arm declarations are interpreted as specified by the interface pointer attribute. In these contexts, the pointer attributes **ref** and **ptr** may be applied to override the default for the top-level pointer in the pointer declaration.

Pointer Attributes in typedefs

Pointer attributes are allowed in **typedefs**. Pointer declarators at the top level of a **typedef** with a pointer attribute are always treated in accordance with the specified pointer attribute. Pointer declarators at the top level of a **typedef** without a pointer attribute and non-top-level pointers are interpreted according to the <ptr_attr> in the interface header of the interface in which it is defined.

Pointer attributes may only be applied where an explicit pointer declaration occurs. There is no way to override the pointer attribute of a declared type at the reference site.

4.2.20.4 Varying Arrays of Pointers

Varying arrays of pointers bear special attention because there may be more valid elements on the return from the call than there were at the start of the call.

---

2. Since indirection operators associate left to right, as in C, the "first" indirection operator is the rightmost one in the IDL source.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Varying Arrays of ref Pointers

Because a reference pointer must not have the value NULL, special requirements apply to varying or conformant varying arrays of reference pointers. If:

- there is an **in,out** or **out** varying or conformant varying array of reference pointers
- the associated <attr_var> (**last_is**, **first_is** or **length_is**) is also **in,out** or **out**

then all array elements that are valid at the time of the call (the **in, out** case) or may be valid at the time of the return from the call must point to storage that can be written with the returned referents.

Unless the client and server have made some private arrangement, outside the IDL, that is binding on all client and server implementations, the caller must initialise all pointers of the array to point to valid memory, even those outside the limit specified by the **last_is**, **first_is** or **length_is** variable at the time of the call.

On the callee side, the callee stub must instantiate storage for each pointer in the varying array, regardless of the value of the associated **last_is**, **first_is**, or **length_is** variable at the time of the call, provided that the layout of the pointed-to storage can be determined at compile time; that is, the pointed to storage must contain no full pointers, unions, conformant arrays, conformant structures, and so on.

Varying Arrays of ptr Pointers

Varying arrays of full pointers are treated differently. Again, consider the case cited above: an **in,out** or **out** varying or conformant varying array whose associated <attr_var> (**last_is**, **first_is** or **length_is**) is also **in, out** or **out**.

When the array elements are full pointers, array elements past the limit established by the associated **last_is**, **first_is** or **length_is** variable need not be initialised before the call on the caller side.

The callee stub does must initialise the array elements past the limit established by the associated **last_is**, **first_is** or **length_is** variable. If the called user code increases the number of valid elements, it must initialise those elements before the call returns to the called stub code.

On return from a call, array elements that were not valid at the time of a call are treated as uninitialised.

4.2.20.5 Restrictions on Pointers

The pointer support IDL provides is very powerful and flexible. At times this flexibility collides with other features of IDL, resulting in the following restrictions:

- Binding parameters must not have the **ptr** attribute. This restriction refers to the first parameter position, when that parameter is the type **handle_t** or is a type with the **handle** attribute.
- Context handle parameters must not have the **ptr** attribute.
- Types that are the base type of a pipe must not be or contain a pointer.
- Parameters that are pointers and have only the **out** directional attribute must not have the **ptr** attribute.

  Parameters with only the **out** directional attribute may, however, contain full pointers. Such pointers are not initialised at the time the manager code is called. It is the responsibility of the manager code to initialise these pointers, either to NULL or to point to accessible memory, before returning to the callee stub.
- When using the **transmit_as** attribute with parameters that either are or contain pointers, the transmitted type must not be or contain a pointer type. When using the ACS **represent_as** attribute (see Subclause 4.3.6 on page 244) with parameters that either are or contain pointers, the network type must not be or contain a pointer type.
- A parameter or structure member that is referenced by an <attr_var> must not have a **ptr** attribute. (This guarantees that the variable determining array size will never be null.)

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.2.21 Pointers as Arrays

When declaring a conformant array parameter, IDL provides an alternative to using **[]** (brackets). A parameter that is a pointer to a type is treated as an array of that type if the parameter has either of the array attributes **max_is** or **size_is**.

Note that this equivalence of arrays and pointers is only true in parameter lists. As structure or union members, arrays and pointers are distinct. A structure or union member declared with bracket notation declares an array contained within the structure. A member declared to be a pointer is a pointer to a data element. If a structure field is a pointer and has the **size_is** or **max_is** attribute, then it is a pointer to an array of data.

Because of parsing ambiguities, the language does not allow mixing pointer and bracket notation when declaring a pointer to a conformant array. The language does not allow declaring a pointer to a varying array. A pointer to a structure that contains a varying array must be used instead.

The array attributes controlling the valid range of elements may be applied to arrays declared as pointers, just as they apply to arrays declared with brackets; the **size_is**, **max_is**, **last_is** and **length_is** attributes may be applied to pointers, just as they may be to arrays.

4.2.21.1 Pointers with the string Attribute

A pointer to any of the base types specified in Subclause 4.2.9 on page 225 may have the **string** attribute. Its meaning is that the pointer is a pointer to a string of the base type. Such a string is equivalent to a conformant array, and is treated in accordance with the rules for a conformant array with the **string** attribute.

4.2.21.2 Possible Ambiguity Resolved

When dealing with an operation such as:

```
void op ([in] long s, [in, size_is(s)] long * * myarray);
```

a possible ambiguity arises. Is **myarray** a pointer to an array of **long**s, or an array of pointers to **long**s? IDL defines this signature to be an array of pointers to **long**s. The **max_is** and **size_is** attributes always apply to the top-level, or rightmost, * (asterisk) in the IDL signature of the parameter.

4.2.22 Operations

The syntax of an operation declaration is as follows:

```
<op_declarator> ::= [ <operation_attributes> ] <simple_type_spec>
       <Identifier> <parameter_declarators>
```

The syntax for operation attributes is:

```
<operation_attributes> ::= <[>  <operation_attribute>
       [ , <operation_attribute> ] ... <]>
<operation_attribute> ::= idempotent
       | broadcast
       | maybe
       | <usage_attribute>
       | <ptr_attr>
```

If none of **idempotent**, **broadcast**, **maybe** is present, at-most-once semantics are applied; the operation is executed no more than one time. If a remote call fails, it is not be retried automatically if there is any chance that the called code has started execution.

When used as an `<operation_attribute>`, the `<ptr_attr>` must not be the **ref** attribute. (See **Pointer Attributes on Function Results** on page 235 for more information on `<ptr_attr>` as an **<operation_attribute>**.)

An operation must not have an array type result.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.22.1 The idempotent Attribute

The **idempotent** attribute indicates that the operation does not modify any state and/or yields the same result on each invocation.

An operation with the **idempotent** attribute must not have any pipe parameters.

4.2.22.2 The broadcast Attribute

The **broadcast** attribute specifies that the operation may be invoked multiple times concurrently as the result of a single RPC. This is different from the **idempotent** attribute, which specifies that a call may be retried in the event of failure. Operations with the **broadcast** attribute may be sent to multiple servers, effectively concurrently. The output arguments that the caller receives are taken from the first reply to return successfully. An operation with the **broadcast** attribute is implicitly an idempotent operation.

An operation with the **broadcast** attribute must not have any pipe parameters.

4.2.22.3 The maybe Attribute

The **maybe** attribute specifies that the operation's caller must not require and must not receive a response or fault indication. An operation with the **maybe** attribute must not contain any output parameters and is implicitly an idempotent operation.

An operation with the **maybe** attribute must not have any pipe parameters.

4.2.23 Parameter Declarations

The following subclauses describe the syntax and semantics of parameter declarations.

4.2.23.1 Syntax

The syntax rules for parameter declarations are as follows:

```
<param_declarators> ::= ( [ <param_declarator>
        [ , <param_declarator> ] ... ] )
        | ( void )
<param_declarator> ::= <param_attributes> <type_spec> <declarator>
<param_attributes> ::= <[> <param_attribute>
        [ , <param_attribute> ] ... <]>

<param_attribute> ::= <directional_attribute>
        | <field_attribute>
<directional_attribute> ::= in
        | out
```

4.2.23.2 Semantics and Restrictions

If the `<param_declarators>` production consists of only a pair of parentheses, the semantics are the same as if the keyword **void** appeared between the parentheses. The remote operation has no parameters.

4.2.23.3 Directional Attributes

At least one directional attribute must be specified for each parameter. Table 4-2 gives the meanings of the directional attributes.

Table 4-2 IDL Directional Attributes

Attribute	Meaning
in	The parameter is passed from the caller to the callee.
out	The parameter is passed from the callee to the caller.

A parameter with the **out** attribute must be either an array or an explicitly declared pointer. An explicitly declared pointer is declared by a `pointer_declarator`, rather than by a `simple_declarator` with a named parameter as its `type_specifier`.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Any parameter from which the binding for a call is to be derived must have the **in** attribute. If an operation has an **in handle_t**, **in** customised handle (**handle**) or **in** context handle (**context_handle**), and also contains an **in,out** context handle, the **in,out** context handle may be null. If an operation does not have an **in handle_t**, **in** customised handle or **in** context handle, but does contain one or more **in,out** context handles, at least one of the **in,out** context handles must be non-null.

4.2.23.4 Aliasing in Parameter Lists

If two pointer parameters in a parameter list point at the same data item, or at data structures that have some items in common, parameter **aliasing** is said to occur.

Aliasing is supported only for full pointers; both parameters involved in the aliasing must be full pointers. Full pointers are aliases when:

a) They have the same value.

b) They are declared to point to the same type.

c) The size of the pointed-to type, as determined at or before run time, is the same.

The scope of aliasing is a single instance of an RPC call. Aliasing is not preserved across nested RPCs.

Aliasing of reference pointers is not supported and yields unspecified results. Aliasing is not supported for the case where the target of one pointer is a component (for example, a structure field or array element) of another pointer. This yields unspecified results.

4.2.24 Predefined Types

Predefined types are data types derived from the base types that are intrinsic to the IDL language. The syntax for predefined types is as follows:

```
<predefined_type_spec> ::= error_status_t
       | <international_character_type>
<international_character_type> ::= ISO_LATIN_1
        | ISO_MULTILINGUAL
        | ISO_UCS
```

4.2.25 The error_status_t Type

The **error_status_t** type is used to declare an object in which communications and fault status information can be held. This is the appropriate type for objects with the **comm_status** or **fault_status** attributes.

Note: The **error_status_t** type is transmitted as an IDL **unsigned long**. However, implementations may choose to unmarshal this type as a local operating system error type instead of **unsigned long**.

4.2.26 International Character Types

The following types may be used to represent alternative character sets:

ISO_LATIN_1
ISO_MULTI_LINGUAL
ISO_UCS

Data of type **char** is subject to ASCII/EBCDIC conversion when it is transmitted by the RPC mechanism. The predefined international character types are protected from data representation format conversion.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.27 Anonymous Types

An enumeration type is said to be **anonymous** if it is not a type named through a **typedef** statement. A structure or union type is said to be anonymous if it does not have a tag and is not named through a **typedef** statement. The following rules apply to the usage of anonymous types:

- A parameter cannot have an anonymous type.
- A function result cannot have an anonymous type.
- The target of a pointer cannot have an anonymous type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3 The Attribute Configuration Source

The Attribute Configuration Source (ACS) is used to specify details of a stub to be generated. ACS is used to create a separate attribute configuration source that accompanies an IDL specification. The ACS specification affects the interface between the application code and the stub; for example, it can specify whether an explicit binding handle parameter is used. The ACS specification also affects the way stub code is generated; for example, it can specify in-line or out-of-line data marshalling. The ACS specification does not affect the way the data is transmitted or received during a remote procedure call; this is determined entirely by the IDL specification.

4.3.1 Comments

Comments in ACS conform to the same rules as IDL comments.

4.3.2 Identifiers

Each ACS source is associated with some IDL source. The following associations apply:

- The interface name in the ACS source must be the same as the interface name in the interface definition.
- Any type names used in the ACS source must have been declared as names of types in the interface definition, except for type names that are `<ACS_repr_type>`s or a type used in the `<ACS_implicit_handle_attr>`.
- Any operation names used in the ACS source must have been declared as names of operations in the interface definition.
- If an identifier occurs as a parameter name within an operation declaration in the ACS source, that identifier must have been used as the name of a parameter within the declaration of the corresponding operation in the interface definition. Not all such parameters need occur in the ACS operation definition.
- If a type is declared in a **typedef** with ACS attributes, those attributes are not inherited by other types declared using the type with ACS attributes.

  Note: This is the opposite of the case for IDL type attributes, which are inherited. (See Subclause 4.2.23.2 on page 238.)

4.3.3 Syntax

The syntax of an ACS specification is as follows:

```
<ACS_interface> ::= <ACS_interface_header>  { <ACS_interface_body> }
where
<ACS_interface_header> ::=  [ <ACS_interface_attr_list> ] interface
      <ACS_interface_name>
<ACS_interface_attr_list> ::= <[> <ACS_interface_attrs> <]>
<ACS_interface_attrs> ::= <ACS_interface_attr> [ , <ACS_interface_attr> ] ...
<ACS_interface_attr> ::= <ACS_code_attr>
       | <ACS_nocode_attr>
       | <ACS_inline_attr>
       | <ACS_outofline_attr>
       | <ACS_explicit_handle_attr>
       | <ACS_implicit_handle_attr>
       | <ACS_auto_handle_attr>
<ACS_explicit_handle_attr> ::= explicit_handle
<ACS_implicit_handle_attr> ::= implicit_handle ( <ACS_named_type>
      <Identifier> )
<ACS_auto_handle_attr> ::= auto_handle
<ACS_interface_name> ::= <Identifier>
<ACS_interface_body> ::= [ <ACS_body_element> ] ...
<ACS_body_element> ::= <ACS_include> ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        | <ACS_type_declaration> ;
        | <ACS_oper> ;
<ACS_include> ::= include <ACS_include_list>
<ACS_include_list> ::= <ACS_include_name> [ , <ACS_include_name> ] ...
<ACS_include_name> ::= "<Import_string>"
<ACS_type_declaration> ::=  typedef [ <ACS_type_attr_list> ] <ACS_named_type>
<ACS_named_type> ::= <Identifier>
<ACS_type_attr_list> ::= <[> <ACS_type_attrs> <]>
<ACS_type_attrs> ::= <ACS_type_attr> [ , <ACS_type_attr> ] ...
<ACS_type_attr> ::= <ACS_repr_attr>
        | <ACS_inline_attr>
        | <ACS_outofline_attr>
        | <ACS_heap_attr>
<ACS_repr_attr> ::= represent_as ( <ACS_repr_type> )
<ACS_repr_type> ::= <Identifier>
<ACS_oper> ::= [ <ACS_oper_attr_list> ] <Identifier> ( [ <ACS_params> ] )
<ACS_oper_attr_list> ::= <[> <ACS_oper_attrs> <]>
<ACS_oper_attrs> ::= <ACS_oper_attr> [ , <ACS_oper_attr> ] ...
<ACS_oper_attr> ::= <ACS_commstat_attr>
        | <ACS_faultstat_attr>
        | <ACS_code_attr>
        | <ACS_nocode_attr>
        | <ACS_explicit_handle_attr>
        | <ACS_enable_alloc_attr>
<ACS_params> ::= <ACS_param> [ , <ACS_param> ] ...
<ACS_param> ::= [ <ACS_param_attr_list> ] <Identifier>
<ACS_param_attr_list> ::= <[> <ACS_param_attrs> <]>
<ACS_param_attrs> ::= <ACS_param_attr> [ , <ACS_param_attr> ] ...
<ACS_param_attr> ::= <ACS_commstat_attr>
        | <ACS_faultstat_attr>
        | <ACS_heap_attr>
<ACS_code_attr> ::= code
<ACS_nocode_attr> ::= nocode
<ACS_inline_attr> ::= in_line
<ACS_outofline_attr> ::= out_of_line
<ACS_commstat_attr> ::= comm_status
<ACS_faultstat_attr> ::= fault_status
<ACS_heap_attr> ::= heap
<ACS_enable_alloc_attr> ::= enable_allocate
```

4.3.4 Include Declaration

The **include** statement specifies one or more header sources that are are included in generated stub code, for example, via C-preprocessor **#include** statements. The user must supply the header sources. A C-language definition source (**.h** file) must always be provided for use when compiling the stubs. If the stubs are being generated for another language, then a definition source for that language must be provided as well. The same ACS source may not work with different target languages.

If the include statements do not fully specify the location of the header sources, the compiler uses implementation-specific searching mechanisms to locate them. Similarly, if the file extension is not specified, then the compiler appends the appropriate file extension for the language of choice.

Include sources are necessary if, in the ACS source, there occur one or more types specified in the **represent_as** clause's <ACS_named_type> or the **implicit_handle** attribute's <type_spec> that are not types defined in the interface definition (or any imported interfaces). Since the definitions of such types are needed by the generated stub code, the user must supply them in this manner.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3.5 Specifying Binding Handles

The means of providing binding information to the RPC run-time system is the binding handle. Binding handles (or simply, handles) may be passed as parameters of the operation or fetched by the generated stub from a static area. A handle passed as an operation parameter is termed an **explicit handle**. If an explicit handle is used, it is always the first parameter of an operation. Explicit handles may be declared in the IDL source, in which case both the client and server must use the explicit handle. Explicit handles may also be declared in the ACS source, in which case the two sides (client and server) may make the decision to use an explicit handle separately.

When an interface contains one or more operations whose first parameter is not an explicit handle, and which do not have an **in** or **in,out** context handle, a means of providing a handle is needed. The **implicit_handle** and **auto_handle** attributes provide this capability.

If an interface has an operation requiring an implicit handle, and no ACS source is supplied, or the supplied ACS source does not specify either **implicit_handle** or **auto_handle**, then the default **auto_handle** attribute is applied. The **auto_handle** attribute is also applied in the event that a operation has an **in,out** context handle but no other binding handle mechanism.

4.3.5.1 The explicit_handle Attribute

When used as an interface attribute, the **explicit_handle** attribute is applied to all operations in the interface. When used as an operation attribute, it is applied to only the specified operation.

The **explicit_handle** attribute specifies that the operation has an additional first parameter of type **handle_t** and named **IDL_handle**, even if one is not explicitly declared in the IDL source. Customised binding handles must be declared in the IDL source. The **explicit_handle** attribute may occur as an interface attribute only if the **implicit_handle** attribute and **auto_handle** attribute do not occur.

4.3.5.2 The implicit_handle Attribute

The **implicit_handle** attribute is one of the methods for specifying handles. Under the **implicit_handle** method, the handle used on calls to operations without a handle in the first parameter position is the data object specified in the **implicit_handle** attribute.

The **implicit_handle** attribute must occur at most once in the ACS source. The **implicit_handle** attribute may occur only if the **auto_handle** attribute does not occur and the **explicit_handle** attribute does not occur as an interface attribute.

The `<type_spec>` specified in the **implicit_handle** attribute need not be specified in the associated interface definition source. If it is specified, then the definition specified in the IDL source is used; it must be either a type that resolves to the type **handle_t** or a type with the **handle** attribute. If it is not a type defined in the interface definition source, then the ACS source must contain an include statement, specifying a definition source that defines the `<type_spec>`. The type is treated as a customised handle; that is, as if it had the **handle** attribute applied to it.

4.3.5.3 The auto_handle Attribute

The **auto_handle** attribute indicates that any operations needing handles are automatically bound; that is, a client that makes a call on that operation makes no specification as to which server the operation may execute on.

The **auto_handle** attribute must occur at most once.

The **auto_handle** attribute may occur only if the **implicit_handle** attribute does not occur and the **explicit_handle** attribute does not occur as an interface attribute.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3.6 The represent_as Attribute

The **represent_as** attribute associates a named local type in the target language (<ACS_repr_type>) with a transfer type (<ACS_named_type>) that is transferred between caller and callee.

There are some restrictions on the types to which the **represent_as** attribute may be applied. The following types must not have the **represent_as** attribute:

- pipe types
- types used as the base type in a pipe definition
- conformant, varying or conformant varying arrays
- structures whose last member is a conformant array (a conformant structure)
- pointers or types that contain a pointer.

4.3.7 The code and nocode Attributes

At most, one of the **code** and **nocode** attributes may appear in the interface attribute list. If neither is present, the effect is as if **code** is present.

The **nocode** attribute is only honoured when generating a client stub. Servers must support all defined operations.

If **code** appears in the interface attribute list, stub code is be generated for any operation in the interface that does not appear in the ACS source with **nocode** in its operation attribute list.

If **nocode** appears in the interface attribute list, stub code is only generated for those operations in the interface that appear in the ACS source with **code** in their operation attribute lists.

At most, one of **code** and **nocode** may appear in an operation attribute list. If both the interface and the operation have a **code** or **nocode** attribute, the attribute applied to the operation overrides the attribute applied to the interface.

4.3.8 The in_line and out_of_line Attributes

The **in_line** and **out_of_line** attributes may be applied at the interface or type levels. At most, one may be applied to any interface or type. If the **out_of_line** attribute is specified on a type that is not a candidate for out-of-line marshalling, it is ignored.

If neither of these attributes is specified in the interface attribute list, the effect is as if the **in_line** attribute is specified. If neither of these attributes is specified for a type, then the effect is as if the attribute specified in the interface attribute list is specified for that type. The types that are candidates for out-of-line marshalling are as follows:

- structures
- unions
- arrays
- context handles
- pipes.

If a candidate type has the **out_of_line** attribute, marshalling and/or unmarshalling of that type is performed through a subroutine call. Otherwise the marshalling and/or unmarshalling may be performed by code that is part of the direct control flow in the stubs for any operations that have a parameter of that type. When dealing with types imported from another interface definition module, it is the ACS source associated with the imported module that determines whether types defined in that module are marshaled in-line or out-of-line.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3.9 Return Statuses

Two attributes, **comm_status** and **fault_status**, are available to provide a status return mechanism for certain error conditions that occur during the execution of remote routines. Portable applications must include an ACS specification that specifies these attributes.

4.3.9.1 The comm_status Attribute

The **comm_status** attribute must occur at most once per operation. It may appear as an operation attribute for the operation, or as a parameter attribute for one of the parameters of the operation.

If the **comm_status** attribute appears as an operation attribute, the operation must have been defined to deliver a result of type **error_status_t**. If the run-time system detects that some communications error (for example, a broken connection or a timeout) has occurred during execution of the operation, the error code is returned as the operation result. If the run-time system does not detect a communications failure, then the operation result has the value returned by the manager routine.

If the **comm_status** attribute appears as a parameter attribute, the `<Identifier>` associated with it need not be the `<Identifier>` of a parameter defined in the IDL. If the **comm_status** attribute does specify the `<Identifier>` of a parameter defined in the IDL, then the parameter must be an **out** parameter of type **error_status_t***. In the event that the remote call completes successfully, the parameter has the value assigned by the called procedure.

If the `<Identifier>` associated with the **comm_status** attribute is not the `<Identifier>` of a parameter defined in the IDL, then an extra **out** parameter of type **error_status_t*** is created. This follows the last parameter defined in the IDL, unless a parameter with the **fault_status** attribute is present. If a parameter with the **fault_status** attribute is present, then the parameters are defined in the order they appear in the ACS. In the case of successful completion of the call, the extra parameter has the value **error_status_ok**.

If a communications error occurs during execution of the operation, the error code is returned in the parameter with the **comm_status** attribute.

For a summary of which errors are reported through the **comm_status** mechanism when enabled, refer to Annex E.

4.3.9.2 The fault_status Attribute

The **fault_status** attribute is similar to the **comm_status** attribute. However, it deals with certain failures of the remote routine rather than communications errors.

The **fault_status** attribute must occur at most once per operation. It may appear as an operation attribute for the operation, or as a parameter attribute for one of the parameters of the operation.

If the **fault_status** attribute appears as an operation attribute, the operation must have been defined to deliver a result of type **error_status_t**. If the remote procedure fails in a way that causes a **fault** PDU to be returned, the error code is returned as the operation result. If a failure is not detected, then the operation result has the value returned by the manager routine.

If the **fault_status** attribute appears as a parameter attribute, the `<Identifier>` associated with it need not be the `<Identifier>` of a parameter defined in the IDL. If the **fault_status** attribute does specify the `<Identifier>` of a parameter defined in the IDL, then the parameter must be an **out** parameter of type **error_status_t***. In the event that the remote call completes successfully, the parameter has the value assigned by the called procedure.

If the `<Identifier>` associated with the **fault_status** attribute is not the `<Identifier>` of a parameter defined in the IDL, then an extra **out** parameter of type **error_status_t*** is created. This follows the last parameter defined in the IDL unless a parameter with the **com_status** attribute is present. If a parameter with the **com_status** attribute is present, then the parameters are defined in the order in which they appear in the ACS. In the case of successful completion of the call, the extra parameter has the value **error_status_ok**.

If a suitable error occurs during execution of the operation, the error code is returned in the parameter with the **fault_status** attribute.

For a summary of which exceptions are reported through the **fault_status** mechanism when enabled, refer

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

to Annex E.

4.3.9.3 Interaction of the comm_status and fault_status Attributes

It is possible for one operation to have both the **fault_status** and the **comm_status** attributes, either as operation attributes or as parameter attributes.

If both attributes are applied as operation attributes, or both attributes are applied to the same parameter, then the operation or parameter has the value **error_status_ok** if no error occurred. Otherwise, it has the appropriate **comm_status** or **fault_status** value. Since the values returned in parameters with the **comm_status** attribute are disjoint from the values returned in parameters with the **fault_status** attribute, and it is not possible for a single call to result in two failures, there is no ambiguity in interpreting the returned status value.

If both attributes are specified for an operation, and each attribute refers to an `<Identifier>` that is not defined in the IDL source, then two extra parameters are defined after the last parameter defined in the IDL source. These parameters are defined in the order in which they appear in the ACS source.

4.3.10 The heap Attribute

The **heap** attribute specifies that the server copy of the parameter or parameter of the type so specified will always be allocated in heap memory, rather than on the stack.

4.3.11 The enable_allocate Attribute

The **enable_allocate** attribute causes the stub to initialise stub memory management in conditions where it otherwise would not do so. It has no effect if the stub would perform the initialisation for other reasons.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.4 IDL Grammar Synopsis

The following subclauses give a synopsis of the IDL grammar for quick reference.

4.4.1 Grammar Synopsis

This subclause provides a synopsis of the IDL productions discussed in Section 4.

```
(1) <interface> ::= <interface_header>  { <interface_body> }
(2) <interface_header> ::= <[>  <interface_attributes> <]> interface
      <Identifier>
(3) <interface_attributes> ::= <interface_attribute>
      [ , <interface_attribute> ] ...
(4) <interface_attribute> ::= uuid ( <Uuid_rep> )
      | version ( <Integer_literal> )
      | endpoint ( <port_spec> [ , <port_spec> ] ... )
      | local
      | pointer_default ( <ptr_attr> )
(5) <port_spec> ::= " <Family_string> :  <[> <Port_string> <]> "
(6) <interface_body> ::= [ <import> ... ] <interface_component>
      [ <interface_component> ... ]
(7) <import> ::= import <import_list> ;
(8) <interface_component> ::= <export> | <op_declarator> ;
(9) <export> ::= <type_declarator> ;
      | <const_declarator> ;
      | <tagged_declarator> ;
(10) <import_list> ::= <import_name> [ , <import_name> ] ...
(11) <import_name> ::=  "<Import_string>"
(12) <const_declarator> ::= const <const_type_spec> <Identifier> = <const_exp>
(13) <const_type_spec> ::= <primitive_integer_type>
      | char
      | boolean
      | void *
      | char *

(14) <const_exp> ::= <Integer_const_exp>
      | <Identifier>
      | <string>
      | <character_constant>
      | NULL
      | TRUE
      | FALSE
(14.01) <integer_const_exp> ::= <conditional_exp>
(14.02) <conditional_exp> ::= <logical_or_exp>
      | <logical_or_exp> ? <integer_const_exp> : <conditional_exp>
(14.03) <logical_or_exp> ::= <logical_and_exp>
      | <logical_or_exp> <||> <logical_and_exp>
(14.04) <logical_and_exp> ::= <inclusive_or_exp>
      | <logical_and_exp> && <inclusive_or_exp>
(14.05) <inclusive_or_exp> ::= <exclusive_or_exp>
      | <inclusive_or_exp> <|> <exclusive_or_exp>
(14.06) <exclusive_or_exp> ::= <and_exp>
      | <exclusive_or_exp> ^ <and_exp>
(14.07) <and_exp> ::= <equality_exp>
      | <and_exp> & <equality_exp>
(14.08) <equality_exp> ::= <relational_exp>
      | <equality_exp> == <relational_exp>
      | <equality_exp> != <relational_exp>
(14.09) <relational_exp> ::= <shift_exp>
      | <relational_exp> <<> <shift_exp>
      | <relational_exp> <>> <shift_exp>
      | <relational_exp> <<=> <shift_exp>
      | <relational_exp> <>=> <shift_exp>
(14.10) <shift_exp> ::= <additive_exp>
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
      | <shift_exp> <<<> <additive_exp>
      | <shift_exp> <>>> <additive_exp>
(14.11) <additive_exp> ::= <multiplicative_exp>
      | <additive_exp> + <multiplicative_exp>
      | <additive_exp> - <multiplicative_exp>

(14.12) <multiplicative_exp> ::= <unary_exp>
      | <multiplicative_exp> * <unary_exp>
      | <multiplicative_exp> / <unary_exp>
      | <multiplicative_exp> % <unary_exp>
(14.13) <unary_exp> ::= <primary_exp>
      | + <primary_exp>
      | - <primary_exp>
      | ~ <primary_exp>
      | ! <primary_exp>
(14.14) <primary_exp> ::= <Integer_literal>
      | <Identifier>
(15) <string> ::= "[<Character>] ... "
(16) <character_constant> ::= '<Character>'
(17) <type_declarator> ::= typedef [ <type_attribute_list> ] <type_spec>
     <declarators>
(18) <type_attribute_list> ::= <[> <type_attribute>
     [ , <type_attribute> ] ... <]>
(19) <type_spec> ::= <simple_type_spec>
     | <constructed_type_spec>
(20) <simple_type_spec> ::= <base_type_spec>
     | <predefined_type_spec>
     | <Identifier>
(21) <declarators> ::= <declarator> [ , <declarator> ] ...
(23) <declarator> ::=  <pointer_opt> <direct_declarator>
(24) <direct_declarator> ::= <Identifier>
     | ( <declarator> )
     | <array_declarator>
     | <function_declarator>
(26) <tagged_declarator> ::= <tagged_struct_declarator>
     | <tagged_union_declarator>
(27) <base_type_spec> ::= <floating_pt_type>
     | <integer_type>
     | <char_type>
     | <boolean_type>
     | <byte_type>
     | <void_type>
     | <handle_type>
(28) <floating_pt_type> ::= float
     | double
(29) <integer_type> ::= <primitive_integer_type>
     | hyper [unsigned] [int]
     | unsigned hyper [int]
(29.1) <primitive_integer_type> ::= <signed_integer>
     | <unsigned_integer>
(30) <signed_integer> ::= <integer_size> [ int ]
(31) <unsigned_integer> ::= <integer_size> unsigned [ int ]
     | unsigned <integer_size> [ int ]
(32) <integer_size> ::=  long
     | short
     | small
(33) <char_type> ::= [ unsigned ] char
(34) <boolean_type> ::= boolean
(35) <byte_type> ::= byte
(36) <void_type> ::= void
(37) <handle_type> ::= handle_t
(38) <constructed_type_spec> ::= <struct_type>
     | <union_type>
     | <enumeration_type>
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
      | <tagged_declarator>
      | <pipe_type>
(39) <tagged_struct_declarator> ::= struct <tag>
      | <tagged_struct>
(40) <struct_type> ::= struct  { <member_list> }
(41) <tagged_struct> ::= struct <tag> { <member_list> }
(42) <tag> ::= <Identifier>

(43) <member_list> ::= <member> [ <member> ] ...
(44) <member> ::= <field_declarator> ;
(45) <field_declarator> ::= [ <field_attribute_list> ] <type_spec>
     <declarators>
(46) <field_attribute_list> ::= <[> <field_attribute>
     [ , <field_attribute> ] ... <]>
(47) <tagged_union_declarator> ::= union <tag>
     | <tagged_union>
(48) <union_type> ::= union <union_switch> { <union_body> }
     | union { <union_body_n_e> }
(48.1) <union_switch> ::= switch ( <switch_type_spec> <Identifier> )
     [ <union_name> ]
(49) <switch_type_spec> ::= <primitive_integer_type>
      | <char_type>
      | <boolean_type>
      | <Identifier>
(50) <tagged_union> ::= union <tag> <union_switch> { <union_body> }
      | union <tag> { <union_body_n_e> }
(51) <union_name> ::= <Identifier>
(52) <union_body> ::= <union_case> [  <union_case> ] ...
(52.1) <union_body_n_e> ::=  <union_case_n_e> [ <union_case_n_e> ] ...
(53) <union_case> ::= <union_case_label> [ <union_case_label> ] ...
     <union_arm>
      | <default_case>
(53.1) <union_case_n_e> ::= <union_case_label_n_e> <union_arm>
      | <default_case_n_e>
(54) <union_case_label> ::= case <const_exp> :
(54.1) <union_case_label_n_e> ::= <[> case (<const_exp>
      [ , <const_exp> ] ...) <]>
(55) <default_case> ::= default : <union_arm>
(55.1) <default_case_n_e> ::= <[> default <]> <union_arm>
(55.2) <union_arm> ::=  [ <field_declarator> ] ;
(55.3) <union_type_switch_attr> ::= switch_type ( <switch_type_spec> )
(55.4) <union_instance_switch_attr> ::=  switch_is ( <attr_var> )
(57) <enumeration_type> ::= enum { <Identifier> [ , <Identifier> ] ... }
(58) <pipe_type> ::= pipe <type_spec>
(59) <array_declarator> ::= <direct_declarator> <array_bounds_declarator>
(61) <array_bounds_declarator> ::= <[> [ <array_bound> ] <]>
      | <[> <array_bounds_pair> <]>
(62) <array_bounds_pair> ::= <array_bound> .. <array_bound>
(63) <array_bound> ::= *
      | <integer_const_expr>
      | <Identifier>
(64) <type_attribute> ::= transmit_as ( <xmit_type> )
      | handle
      | align ( <integer_size> )
      | <usage_attribute>
      | <union_type_switch_attr>
      | <ptr_attr>
(65) <usage_attribute> ::= string
      | context_handle
(66) <xmit_type> ::= <simple_type_spec>
(67) <field_attribute> ::= first_is ( <attr_var_list> )
      | last_is ( <attr_var_list> )
      | length_is ( <attr_var_list> )
      | max_is ( <attr_var_list> )
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
       | size_is ( <attr_var_list> )
       | <usage_attribute>
       | <union_instance_switch_attr>
       | ignore
       | <ptr_attr>
(68) <attr_var_list> ::= <attr_var> [ , <attr_var> ] ...
(69) <attr_var> ::= [ [ * ] <Identifier> ]
(70) <pointer_opt> ::= [<pointer>]
(70.1) <pointer> ::= *...
(70.2) <ptr_attr> ::= ref
       | full
(71) <op_declarator> ::=  [ <operation_attributes> ]
      <simple_type_spec> <Identifier> <parameter_declarators>
(72) <operation_attributes> ::= <[>  <operation_attribute>
       [ , <operation_attribute> ] ... <]>
(73) <operation_attribute> ::= idempotent
       | broadcast
       | maybe
       | <usage_attribute>
       | <ptr_attr>
(74) <param_declarators> ::= ( [ <param_declarator> [ ,
      <param_declarator> ] ... ] )
       | ( void )
(75) <param_declarator> ::= <param_attributes> <type_spec> <declarator>
(76) <param_attributes> ::= <[> <param_attribute> [ ,
      <param_attribute> ] ... <]>
(77) <param_attribute> ::= <directional_attribute>
       | <field_attribute>
(78) <directional_attribute> ::= in
       | out
(79) <function_declarator> ::= <direct_declarator> <param_declarators>
(80) <predefined_type_spec> ::= error_status_t
       | <international_character_type>
```

4.4.2 Alphabetic Listing of Productions

Table 4-2 lists the terminals and non-terminals of the grammar in alphabetic order, with their numbers and the numbers of all productions that use them.

Table 4-3 Alphabetic Listing of Productions

Production Name	Number	Used In
<additive_exp>	14.11	14.10
<and_exp>	14.07	14.06
<array_bound>	63	61, 62
<array_bounds_declarator>	61	60
<array_bounds_pair>	62	61
<array_declarator>	59	24
<attr_var>	69	55.4, 68
<attr_var_list>	68	67
<base_type_spec>	27	20
<boolean_type>	34	27, 49
<byte_type>	35	27
<char_type>	33	27, 49
<Character>	Terminal	15, 16
<character_constant>	16	14
<conditional_exp>	14.02	14.01
<const_declarator>	12	9

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Production Name	Number	Used In
<const_exp>	14	12, 54, 54.1
<const_type_spec>	13	12
<constructed_type_spec>	38	19
<declarator>	23	21, 24, 75
<declarators>	21	17, 45
<default_case>	55	52
<default_case_n_e>	55.1	52.1
<direct_declarator>	24	23, 59, 79
<directional_attribute>	78	77
<enumeration_type>	57	38
<equality_exp>	14.08	14.07
<exclusive_or_exp>	14.06	14.05
<export>	9	8
<Family_string>	Terminal	5
<field_attribute>	67	46, 77
<field_attribute_list>	46	45
<field_declarator>	45	44, 55.2
<function_declarator>	79	24
<Import_string>	Terminal	11
<floating_pt_type>	28	27
<handle_type>	37	27
<Identifier>	Terminal	2,12,14,14.14,20,24,42,48.1,51,57,59,63,69,70,71,79
<import>	7	6
<import_list>	10	7
<import_name>	11	10
<inclusive_or_exp>	14.05	14.04
<integer_const_exp>	14.01	14, 14.02, 63
<integer_size>	32	30, 31, 64
<Integer_literal>	Terminal	4, 14.14
<integer_type>	29	27
<interface>	1	GOAL
<interface_attribute>	4	3
<interface_attributes>	3	2
<interface_body>	6	1
<interface_component>	8	6
<interface_header>	2	1
<international_character_type>	81	80
<logical_and_exp>	14.04	14.03
<logical_or_exp>	14.03	14.02
<member>	44	43
<member_list>	43	40, 41
<multiplicative_exp>	14.12	14.11
<op_declarator>	71	8
<operation_attribute>	73	72
<operation_attributes>	72	71
<param_attribute>	77	76
<param_attributes>	76	75
<param_declarator>	75	74
<param_declarators>	74	71, 79
<pipe_type>	58	38
<pointer>	70.1	70
<pointer_opt>	70	23

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Production Name	Number	Used In
<port_spec>	5	4
<Port_string>	Terminal	5
<predefined_type_spec>	80	20
<primary_exp>	14.14	14.13
<primitive_integer_type>	29.1	13, 29, 49
<ptr_attr>	70.1	4, 64, 67, 73
<relational_exp>	14.09	14.08
<shift_exp>	14.10	14.09
<signed_integer>	30	29
<simple_type_spec>	20	19, 66, 71, 79
<string>	15	14
<struct_type>	40	38
<switch_type_spec>	49	48.1, 55.3
<tag>	42	39, 41, 47, 50
<tagged_declarator>	26	9, 38
<tagged_struct>	41	39
<tagged_struct_declarator>	39	26
<tagged_union>	50	47
<tagged_union_declarator>	47	26
<type_attribute>	64	18
<type_attribute_list>	18	17
<type_declarator>	17	9
<type_spec>	19	17, 45, 58, 75
<unary_exp>	14.13	14.12
<union_arm>	55.2	53, 53.1, 55
<union_body>	52	48, 50
<union_body_n_e>	52.1	48, 50
<union_case>	53	52
<union_case_n_e>	53.1	52.1
<union_case_label>	54	53
<union_case_label_n_e>	54.1	53.1
<union_instance_switch_attr>	55.4	67
<union_name>	51	48.1
<union_switch>	48.1	48, 50
<union_type>	48	38
<union_type_switch_attr>	55.3	64
<unsigned_integer>	31	29
<usage_attribute>	65	64, 67, 73
<Uuid_rep>	Terminal	4
<void_type>	36	27
<xmit_type>	66	64

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.5 IDL Constructed Identifiers

Table 4-3 lists the various classes of identifiers that are used to construct other identifiers. It shows the various strings that are applied to the identifier to build new identifiers. The table also gives the maximum length that a user-supplied identifier is permitted to have in order for the resulting identifier have 31 characters or less. Identifiers of 31 characters or less fall within the ISO C "minimum maximum" requirement for identifiers. This value need not be meaningful for other target languages. The length specified for interface names assumes a single digit major and minor version number. The actual length allowed for interface names is:

(19 –((**digits-in-major-version**) + (**digits-in-minor-version**)))

Table 4-4 Constructed Identifier Classes

Class of ID	Constructed IDs	Max Length
Interface name	<interface>_v<major_version>_<minor_version>_**c_ifspec** <interface>_v<major_version>_<minor_version>_**s_ifspec** <interface>_v<major_version>_<minor_version>_**epv_t** <interface>_v<major_version>_<minor_version>_**c_epv** <interface>_v<major_version>_<minor_version>_**s_epv**	17
Type with **transmit_as** attribute	<type_id>_**to_xmit** <type_id>_**from_xmit** <type_id>_**free_inst** <type_id>_**free_xmit**	21
Type with **handle** attribute	<type_id>_**bind** <type_id>_**unbind**	24
Type with **context_handle** attribute	<type_id>_**rundown**	23
Pointed-to node type	<type_id>_**mc** <type_id>_**ms** <type_id>_**uc** <type_id>_**us**	28
Type with **out_of_line** attribute	<type_id>**Omr**, <type_id>**Pmr** <type_id>**Ome**, <type_id>**Pme** <type_id>**Our**, <type_id>**Pur** <type_id>**Oue**, <type_id>**Pue**	28
Pipe type	<type_id>_**h** <type_id>_**I**	29
Type with **represent_as** attribute	<type_id>_**to_local** <type_id>_**from_local** <type_id>_**free_inst** <type_id>_**free_local**	20

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.6 IDL and ACS Reserved Words

Reserved words are keywords of the language that must not be used as user identifiers. IDL reserved words are given in the following list:

boolean	**byte**	**case**	**char**
const	**default**	**double**	**enum**
FALSE	**float**	**handle_t**	**hyper**
import	**int**	**interface**	**long**
NULL	**pipe**	**short**	**small**
struct	**switch**	**TRUE**	**typedef**
union	**unsigned**	**void**	

The following list gives the IDL keywords that are reserved when in the context of an attribute: that is, between **[ ]** (brackets):

align	**broadcast**	**case**	**comm_status**
context_handle	**endpoint**	**first_is**	**handle**
idempotent	**ignore**	**implicit_handle**	**in**
last_is	**length_is**	**local**	**max_is**
maybe	**out**	**ptr**	**pointer_default**
size_is	**string**	**switch_is**	**switch_type**
transmit_as	**uuid**	**version**	

The following are ACS reserved words: **include**, **interface**, **typedef**.

The following list gives the ACS keywords that are reserved when in the context of an attribute: that is, between **[ ]** (brackets):

auto_handle	**code**	**comm_status**	**enable_allocate**
explicit_handle	**fault_status**	**heap**	**implicit_handle**
in_line	**nocode**	**out_of_line**	**represent_as**

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 5. Stubs

The IDL source and any ACS are used to generate client and server stubs for a specified interface. Except for data type and syntax mappings, specified for the C language in Annex F, the mappings of IDL and ACS to stub code for a given language are mostly implementation-dependent. However, there are certain additional portability requirements on the application/stub interface as well as interoperability requirements on stub code. These requirements are specified in the following subclauses.

5.1 The Application/Stub Interface

Applications interact with the stub mainly by calling IDL specified remote interfaces (on the client side) and by implementing managers for these interfaces (on the server side). However, in certain cases, applications need to be aware of specific details of stub code. These include:

- parameter marshalling and memory management
- the default manager EPV
- the interface handle
- the pipe processing interface
- type attribute interfaces
- context handle rundown.

The following subclauses specify these aspects of stub code. Where appropriate, they provide C bindings that portable C applications must adhere to.

5.1.1 Parameters

Parameter semantics depend on the IDL directional attributes as follows:

- The value of a parameter with the **in** attribute is marshalled by the client stub when the client invokes the call. It is unmarshalled by the server stub and passed to the server manager on invocation by the stub.
- The value of a parameter with the **out** attribute is marshalled by the server stub when the server manager routine returns. It is unmarshalled by the client stub and passed to the client when the call returns.
- The value of a parameter with the attributes **in, out** is passed from the client to the server as an **in** parameter and passed from the server to the client as an **out** parameter.

In the event of an abnormal end-of-call, resulting from either an exception condition on the server (which may be reported through **fault_status** parameter) or a communications failure (which may be reported through **comm_status** parameter) the values of **out** parameters are undefined.

5.1.1.1 Parameter Memory Management

RPC attempts to extend local procedure call parameter memory management semantics to a situation in which the calling and called procedure no longer share the same memory space. In effect, parameter memory has to be allocated twice, once on the client side, once on the server side. Stubs do as much of the extra allocation work as possible so that the complexities of parameter allocation are transparent to applications. In some cases, however, applications may have to manage parameter memory in a way that differs from the usual local procedure call semantics. This subclause specifies stub and application requirements on parameter allocation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

For the purposes of memory allocation, three classes of parameter need to be considered:

- non-pointer types
- reference pointers
- full pointers.

For all types, the client application supplies parameters to the client stub, which marshals them for transmission to the server. The client application is entirely responsible for managing the memory occupied by the passed parameters. On the server side, the server stub allocates and frees all memory required for the received parameters themselves.

In the case of the pointer types, however, the application and stubs must manage memory not only for the parameters themselves, but also for the pointed-to nodes. In this case, the memory management requirements depend both on the pointer type and on the parameter's directional attributes.

The rules are as follows.

5.1.1.2 Client-side Allocation

in parameters — For all pointer types, the client application must allocate memory for the pointed-to nodes.

out parameters — For reference pointers, the client application must allocate memory for the pointed-to nodes unless the pointer is part of a data structure created by server manager code. For parameters containing full pointers, the stub allocates memory for the pointed-to nodes.

in,out parameters — For reference pointers, the client application must allocate memory for the pointed-to nodes. For full pointers, on making the call, the client application must allocate memory for the pointed-to node. On return, the stub keeps track of whether each parameter is the original full pointer passed by the client, or a new pointer allocated by the server. If a pointer is unchanged, the returned data overwrites the existing pointed-to node. If a pointer is new, the stub allocates memory for the pointed-to node. When a parameter contains pointers, such as an element in a linked list, the stub keeps track of the chain of references, allocating nodes as necessary.

It is the client application's responsibility to free any memory allocated by the stub for new nodes. Clients can call the routine **rpc_sm_client_free**( ) for this purpose.

If the server deletes or eliminates a reference to a pointed to node, an "orphaned" node may be created on the client side. It is the client application's responsibility to keep track of memory that it has allocated for pointed-to nodes and to deal with any nodes for which the server no longer has references.

5.1.1.3 Server-side Allocation

in parameters — For all pointer types, the stub manages all memory for pointed-to nodes.

out parameters — For reference pointers, the stub allocates memory for the pointed-to nodes as long as the size of the targets can be determined at compile time. When the manager routine is entered, such reference pointers point to valid storage. For parameters that contain full pointers, the server manager code must allocate memory for pointed-to nodes. Servers can call the routine **rpc_sm_allocate**( ) for this purpose.

in,out parameters — For reference pointers, the stub allocates memory for pointed-to nodes if either the size of the pointed to nodes can be determined at compile time or the reference pointers point to values received from the client. When the manager routine is entered, such reference pointers point to valid storage. For full pointers, the stub allocates memory for the original pointed-to nodes. The server manager code must allocate memory if it creates new references. Servers can call the routine **rpc_sm_allocate**( ) for this purpose.

The server stub automatically frees all memory allocated by calls to **rpc_sm_allocate**( ).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.1.4 Aliasing

For both **out** and **in,out** parameters, when full pointers are aliases, according to the rules specified in Subclause 4.2.23.4 on page 239, the stubs maintain the pointed-to objects such that any changes made by the server are reflected to the client for all aliases. The stubs detect and correctly handle aliasing both in the case where an alias exists on initiation of an RPC and in the case where an alias is created by the server.

5.1.2 Default Manager EPVs

The IDL compiler must be able to generate server stubs that contain a default manager EPV, as described in Subclause 3.1 on page 47.

5.1.3 Interface Handle

The stub must declare an interface handle according to the naming conventions specified in Subclause 3.1 on page 47.

5.1.4 Pipes

At the caller user code to caller stub interface and the callee stub to callee user code interface, pipes appear as a simple callback mechanism. The processing of a pipe parameter appears to be synchronous to the user-written code. The mechanism implemented in the RPC stub and run-time system allows these apparent callbacks to happen without a real remote callback; therefore, the mechanism is very efficient.

Pipe processing is subject to the following restrictions:

- If an operation has more than one **in** or **in,out** parameter that is a pipe, those pipes must be completely processed in the **in** direction, in the order in which they appear in the operation's signature, before any **out** pipes are filled.
- If an operation has more than one **out** or **in,out** parameter that is a pipe, those pipes must be completely processed in the **out** direction, in the order in which they appear in the operation's signature, after all **in** pipes are drained.

The processing of pipes, and the definition of complete processing, is discussed later in this subclause.

Pipes are defined as type constructors rather than as type attributes because they are manifested at the programming interface in a non-obvious way.

As an example of a pipe, consider the following IDL fragment:

```
typedef
       pipe element_t pipe_t;
```

The client and server stubs must declare pipe processing functions with the following signatures and semantics:

```
typedef
     struct pipe_t {
         /*
          ** pointer to routine call back to pull the next
          ** chunk from this pipe
          */
          void (*pull) (
               char *state, /* in: pipe's state pointer */
               element_t *buf, /* in: buffer in which to place a chunk */
               idl_ulong_int esize, /* in: buffer size (# of elements) */
               idl_ulong_int *ecount /* out: size of chunk (# of elements) */
          );
         /*
          ** pointer to routine call back to push the next
          ** chunk into this pipe
          */
          void (*push) (
               char *state,/* in: pipe's state pointer */
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
            element_t *buf, /* in: buffer from which to copy chunk */
            idl_ulong_int ecount /* in: size of chunk (# of elements) */
        );
        /*
        ** pointer to allocate callback to get buffer.
        ** Only used on caller side.

        */
        void (*alloc) (
            char *state,/* in: pipe's state pointer */
            idl_ulong_int bsize, /* in: requested size (# of *bytes*) */
            element_t **buf, /* out: pointer to allocated buffer */
            idl_ulong_int *bcount /* out: size of alloc'd buffer (# of
                                                              bytes*)*/
        );
        /* pointer to arbitrary storage used by push, pull and alloc. */
        char *state;
    } pipe_t;
```

The routine **pipe_t.alloc** allocates memory from which pipe data can be marshalled or into which pipe data can be unmarshalled. The parameter **bsize** states the preferred size of this memory in terms of a number of bytes. The parameters **bcount** and **buf** describe the actual memory that is allocated. If the **pipe_t.alloc** routine allocates less memory than is requested, the RPC run-time mechanism will use the smaller amount of memory and make more calls back to the user. However, the allocation routine must return at least enough memory to hold one item of the pipe base type; otherwise, an error condition occurs. If the **pipe_t.alloc** routine allocates more memory than is requested, the excess is not used.

The structure member **pipe_t.state** is intended to give the **alloc**, **push** and **pull** routines a means of working together. It is available for the implementor of the client or manager, as described in the following subclauses. The following list enumerates which of these objects are necessary in each of the four possible cases, and who is responsible for supplying them. The generalisation to **in, out** pipes is obvious.

- An **in** pipe, caller side:
  - The **pipe_t** structure is allocated by the user.
  - The **pipe_t.pull**, **pipe_t.alloc** and **pipe_t.state** fields are initialised by the user.
  - The **pipe_t.state** field is available for use by the implementor of the caller code.
  - The structure is passed as the pipe parameter. The structure is passed either by value or by reference at the IDL writer's choice, as indicated by the signature of the operation containing the pipe parameter.
- An **in** pipe, callee side:
  - The **pipe_t** structure is allocated by the stub.
  - The **pipe_t.pull** and **pipe_t.state** fields are initialised by the stub.
  - The **pipe_t.state** field is reserved for use by the stub.
  - The structure is passed as the pipe parameter either by value or by reference.
- An **out** pipe, callee side:
  - The **pipe_t** structure is allocated by the stub.
  - The **pipe_t.push** and **pipe_t.state** fields are initialised by the stub.
  - The **pipe_t.state** field is reserved for use by the stub.
  - The structure is passed as the pipe parameter either by value or by reference.
- An **out** pipe, caller side:
  - The **pipe_t** structure is allocated by the user.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- The **pipe_t.push**, **pipe_t.alloc** and **pipe_t.state** fields are initialised by the user.
- The **pipe_t.state** field is available for use by the implementor of the caller code.
- The structure is passed as the pipe parameter either by value or by reference.

To describe the processing of a pipe parameter, it is useful to consider separately the three cases of **in**, **out** and **in,out** pipes.

5.1.4.1 Processing of in Pipes

Remember that an **in** pipe transfers data from the client to the server. On the client side, this involves the **pull** and **alloc** routines provided by the application code. On the server side, this involves the **pull** routine provided by the server stub. The client stub must marshal an **in** pipe parameter whose type is **pipe_t**, from the previous example. For the parameter of this type, **pipe**, the client stub executes the following code sequence to get chunks of the pipe for marshalling:

```
#define DESIRED_NUM_ELEMENTS ...
long count;
long num_buf_elem;
long num_buf_bytes;
element_t *buf;


do {
        /*
        ** allocate a buffer into which this pipe will be
        ** copied by the pull routine.
        */
        (*(pipe->alloc))(
                pipe->state,
                DESIRED_NUM_ELEMENTS * sizeof(element_t),
                &buf,
                &num_buf_bytes
        );
        num_buf_elem = num_buf_bytes / sizeof(element_t);
        /*
        ** Copy the data to the allocated buffer.
        */
        (*(pipe->pull)) (
                pipe->state,
                buf,
                num_buf_elem,
                &count
        );
} while (count > 0);
```

Thus, the client application code must supply **pull** and **alloc** routines that work in concert to yield a sequence of pointers to chunks of which only the last is empty. The client stub must not modify the state pointed to by **pipe->state**. Note that to transmit a large amount of information that is already in the proper form in memory (that is, already an array of **element_t**), the **alloc** and **pull** routines may conspire to allocate a buffer that already has the information in it and have the **pull** routine be a null routine.

The server stub must unmarshal chunks of the pipe and yield them in response to calls back from the manager. The manager may need to limit the size of the chunks it receives.

The server stub must contain a generated **pull** routine and private storage pointed to by the state field of the structure representing the pipe. The private storage will be used by the stub to manage the pipe operations. The manager reads from the pipe by making calls of the form:

```
#define DESIRED_NUM_ELEMENTS ...
long count;
element_t buf [DESIRED_NUM_ELEMENTS];

do {
        ...
        (*(pipe->pull)) (
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
            pipe->state,
            &buf,
            DESIRED_NUM_ELEMENTS,
            &count
      );
} while (count > 0);
```

The routine **pipe->pull** must unmarshal, into the manager supplied buffer, an amount of data that is nonzero, but no more data than the buffer can hold. The buffer must be large enough to hold at least one item of the pipe base type; otherwise an error condition occurs. The stub need not guarantee to fill the buffer; the amount of data in the buffer must be specified in the **count** parameter to the **pipe->pull** routine.

No more data in the pipe is signaled to the application code by a zero length chunk. Any further calls by the manager result in an error. Any calls to the **n** + 1th **in** pipe's **pull** routine before the first **n in** pipes have been completely drained also results in an error. Also, if a manager returns to the callee stub before all the pipes in the interface have been drained or filled an error results. This is the requirement referred to previously: pipes must be completely processed in the order they appear in the operation signature. There is no guarantee that the chunking seen by the manager will match the chunking supplied by the client's **pull** routine.

5.1.4.2 Processing of out Pipes

Remember that an **out** pipe transfers data from the server to the client. On the client side, this involves the **pipe->push** and **pipe->alloc** routines provided by the application code. On the server side, this involves the **pipe->push** routine provided by the server stub.

For an **out** pipe, the client stub's role is to unmarshal chunks of the pipe into a buffer and call back to the application passing a reference to the buffer. In order to allow the application code to manage its memory usage, and to avoid possibly unnecessary copying, the client stub must first call back to the application's **pipe->alloc** routine to get a buffer. In some cases, this may relieve the **pipe->push** routine of any work.

The client stub may have to go through more than one (**pipe->alloc**, **pipe->push**) cycle in order to unmarshal data that the server marshalled as a single chunk.

There is no guarantee that the chunking seen by the client will match the chunking supplied by the server's **push** routine. The client stub must execute the following code sequence repeatedly, until there is no more data in the pipe:

```
#define DESIRED_NUM_ELEMENTS ...
long count;
long num_buf_elem;
long num_buf_bytes;
element_t *buf;

while (data_in_pipe) {
      /*
      ** allocate a buffer into which this pipe will
      ** be copied by the push routine.
      */
      (*(pipe->alloc)) (
            pipe->state,
            DESIRED_NUM_ELEMENTS * sizeof(element_t),
            &buf,
            &num_buf_bytes
      );
      ...
      num_buf_elem = num_buf_bytes / sizeof(element_t);
      /*
      ** Copy the data to the allocated buffer.
      */
      (*(pipe->push)) (
            pipe->state,
            buf,
            count
      );
};
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
/*
** Terminate the pipe by sending zero bytes of data.
*/
(*(pipe->push)) (
       pipe->state,
       buf,
       0
);
```

On the server side, the stub must implement a **push** routine, which the manager calls back repeatedly passing it a **pipe->state**, a count and a chunk. This routine must marshal the chunk into the out stream. The manager's last call to this routine must pass it a reference to a zero-length chunk so that the stub routine can terminate the pipe.

As with **in** pipes, the stub enforces well-behaved pipe filling on the part of the manager by generating "pipe filling errors", if necessary. Any **out** pipes must be filled completely and in order, after all input pipes have be drained completely. The manager must call the stub-supplied routines with code similar to the following fragment:

```
#define DESIRED_NUM_ELEMENTS ...
long count;
element_t buf [DESIRED_NUM_ELEMENTS];

while (more_pipe_data) {
       ...
       (*(pipe->push)) (
              pipe->state,
              &buf,
              count
       );
};
(*(pipe->push)) (
       pipe->state,
       &buf,
       0
);
```

5.1.4.3 Processing of in,out Pipes

A pipe parameter may be **in,out**. The application's **pull** routine must reinitialise the pipe state after the pipe has been drained so that it can be used by the **push** routine correctly.

At the caller side, the **pipe_t** structure must be provided by the user code. It must have **pull** (for the **in** direction), **push** (for the **out** direction), **alloc** and **state** initialised. During the last **pull** call, when it returns a 0 (zero) **count** to indicate that the pipe is now drained, it must prepare the state variable to receive **push** calls.

At the callee side, the **pipe_t** structure is provided by the stub. The stub must initialise both routine pointers and perform a state variable turn around, as described here.

5.1.5 IDL and ACS Type Attributes

The following subclauses describe IDL and ACS type attributes as they affect the application/stub interface.

5.1.5.1 The IDL transmit_as Attribute

The **transmit_as** attribute associates a presented type in the target language with an IDL transmitted type (<xmit_type>). The presented type is the type seen by clients and servers. The transmitted type is the type that the stub passes to the run-time system for transmission. The application must supply routines that perform conversions between the presented and transmitted types, and to release memory used to hold the converted data.

Table 5-1 on page 262 lists the routines that the application must supply, where **<type_id>** is the identifier part of the type defined in the statement in which the **transmit_as** attribute occurs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table 5-1 Transmitted Type Routines

Routine Name	Routine Use
<type_id>_to_xmit	Allocates an instance of the transmitted type and converts from the presented type to the transmitted type (used by both caller and callee).
<type_id>_from_xmit	Converts from the transmitted type to the presented type (used by both caller and callee).
<type_id>_free_inst	Frees resources used by the presented type, but not the type itself as it is allocated by the stub (used by callee).
<type_id>_free_xmit	Frees storage returned by **<type_id>_to_xmit** routine (used by both caller and callee).

The signatures of these routines are as follows:

```
void <type_id>_to_xmit   ( <presented_type> *, (<transmitted_type> **) )
void <type_id>_from_xmit  ( (<transmitted_type> *), (<presented_type> *) )
void <type_id>_free_inst  ( <presented_type> * )
void <type_id>_free_xmit  ( (<transmitted_type> *) )
```

If the presented type is composed of one or more pointers, then the application's **<type_id>_from_xmit** routine must allocate the targets of any such pointers. The stub storage release behaviour is as follows.

Suppose that the **transmit_as** attribute appears either on the type of a parameter or on a component of a parameter and that the parameter has the **out** or **in,out** attribute. Then, the **<type_id>_free_inst** routine is called automatically for the data item which has the **transmit_as** attribute.

Suppose that the **transmit_as** attribute appears on the type of a parameter and that the parameter has only the **in** attribute. Then, the **<type_id>_free_inst** routine is called automatically.

Finally, suppose that the **transmit_as** attribute appears on a component of a parameter and that the parameter has only the **in** attribute. Then, the **<type_id>_free_inst** routine is not called automatically for the component; the manager application code must release any resources that the component uses, possibly by explicitly calling the **<type_id>_free_inst** routine.

The **<type_id>_free_xmit** routine frees any storage that has been allocated for the transmitted type by **<type_id>_to_xmit**.

5.1.5.2 The IDL handle Attribute

The **handle** attribute specifies that a type can serve as a customised handle. Customised handles permit the design of handles that are meaningful to an application. The client application must provide binding and unbinding routines to convert between the custom handle type and the primitive handle type, **handle_t**.

A primitive handle must contain object UUID and destination information that is meaningful to the client/server run-time support mechanisms. A customised handle may only be defined in a type declaration. It must not be defined in an operation declaration. When a parameter in the first position is a type with the **handle** attribute, the parameter does double duty. It determines the binding for the call, and it is transmitted to the called procedure as a normal parameter. Types with the **handle** attribute in other than the first parameter position are treated as ordinary parameters; their **handle** attribute is ignored, and they do not contribute to the binding process.

The client application must supply the following routines:

```
handle_t    <type_id>_bind  ( <type_id> custom_handle )
void   <type_id>_unbind  ( <type_id> custom_handle, handle_t )
```

where **<type_id>** is the identifier of the customised handle data type and **custom_handle** represents the formal parameters of the customised handle data type. The routine **<type_id>_bind** must generate and return a primitive binding handle from a customised handle of type **<type_id>**. The client stub must call **<type_id>_bind** before it sends the request, and call **<type_id>_unbind** before it returns to the caller. The **<type_id>_unbind** routine actions are application-specific and may have no effect.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.1.5.3 Interaction of IDL transmit_as and IDL handle Attributes

If a type has both the **transmit_as** and **handle** attributes and the type is used as the first parameter in an operation, the **<type_id>_bind** routine must be invoked before the **<type_id>_to_xmit** routine.

However, a type that includes the **handle** attribute in its definition must not be used, directly or indirectly, in the definition of a type with the **transmit_as** attribute. A type that includes the **transmit_as** attribute in its definition must not be used, directly or indirectly, in the definition of a type with the **handle** attribute. The **handle** attribute is not allowed on a type that contains a **transmit_as** type.

5.1.5.4 The ACS represent_as Attribute

The **represent_as** attribute associates a named local type in the target language (**<ACS_repr_type>**) with a transfer type (**<ACS_named_type>**) that is transferred between caller and callee. The user must supply routines that perform conversions between the local and transfer types, and that release memory used to hold the converted data.

Table 5-2 on page 264 lists the routines that the application must supply, where the **represent_as** attribute has been specified for (**<ACS_named_type>**).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-2 Transferred Type Routines

Routine Name	Routine Use
<ACS_named_type>_from_local	Allocates an instance of the network type and converts from the local type to the network type (used by both caller and callee).
<ACS_named_type>_to_local	Converts from the network type to the local type (used by both caller and callee).
<ACS_named_type>_free_inst	Frees storage instance used for the network type (used by both caller and callee).
<ACS_named_type>_free_local	Frees storage returned by the routine **<ACS_named_type>**_from_local (used by callee).

The signatures of these routines are as follows:

```
void <ACS_named_type>_from_local ( (<ACS_repr_type> *), (<ACS_named_type> **))
void <ACS_named_type>_to_local ( (<ACS_named_type> *), (<ACS_repr_type>P *) )
void <ACS_named_type>_free_inst ( (<ACS_named_type> *) )
void <ACS_named_type>_free_local ( (<ACS_repr_type> *) )
```

Suppose that the **represent_as** attribute is applied either to the type of a parameter or to a component of a parameter and that the parameter has the **out** or **in,out** attribute. Then, the **<ACS_named_type>_free_local** routine is called automatically for the data item that has the type to which the **represent_as** attribute was applied.

Suppose that the **represent_as** attribute is applied to the type of a parameter and that the parameter has only the **in** attribute. Then, the **<ACS_named_type>_free_local** routine is called automatically.

Finally, suppose that the **represent_as** attribute is applied to the type of a component of a parameter and that the parameter has only the **in** attribute. Then, the **<ACS_named_type>_free_local** routine is not called automatically for the component; the manager application code must release any resources that the component uses, possibly by explicitly calling the **<ACS_named_type>_free_local** routine.

5.1.5.5 Interaction of the ACS represent_as Attribute and the IDL handle Attribute

A type must not have both the **handle** and **represent_as** attributes.

5.1.5.6 Interaction of the ACS represent_as Attribute with the IDL transmit_as Attribute

If a type has both the **represent_as** and **transmit_as** attributes, the transformations are applied in the appropriate order: on the transmit side, **<type_id>_from_local** then **<type_id>_to_xmit**; on the receive side, **<type_id>_from_xmit** then **<type_id>_to_local**.

5.1.6 Context Handle Rundown

A context handle is opaque to the caller. However, a caller may distinguish between a null context handle and an active one. A context handle whose value is 0 (zero) is termed a **null context handle** and does not represent any currently saved context. A context handle with any other value is termed an **active context handle** and represents saved context.

When making an RPC that will create saved context, the caller must pass a null context handle. The called procedure may return an active context handle. A context handle parameter with only the **out** attribute (that is, without either the **in** of **in,out** attributes) is interpreted as if it were a null context handle at the time of the call. It is the responsibility of the caller to pass the unmodified context handle back to the server on the next call.

The interpretation of the context handle is totally up to the called procedure. If the caller modifies a context handle in any way other than initialising it to 0 (zero) before its first use, then behaviour is unpredictable. It is the responsibility of the callee to return a null context handle when it is no longer maintaining context on behalf of the caller. For example, if the context handle represents an open file and the call closes the file, the callee must set the context handle to 0 (zero) and return it to the caller. If the callee terminates the context and fails to return a null context handle, then the context rundown routine will be erroneously called when the client exits.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For some contexts a context rundown routine may be required. If communications between the caller and the callee break down while the callee is maintaining context for the caller, the run-time system invokes the context rundown routine to enable the callee to clean up this context. When an interface requires context but does not require a context rundown routine, it is sufficient to use parameters that have the **context_handle** attribute. However, where a rundown routine is required, the user must define a named type that has the **context_handle** attribute.

By making the type definition, the user implicitly specifies the name of the rundown routine for the context. The declaration of a type with the **context_handle** attribute and the name **<context_type_name>** specifies a rundown routine with the name **<context_type_name>_rundown**. A rundown routine takes one parameter, the context handle of the context that is to be run down, and delivers no result.

For example, if the application declares:

```
typedef [context_handle] void *my_context
```

it must supply, in the manager application code, a rundown routine that matches the prototype:

```
void my_context_rundown ( void *context_handle );
```

A context handle is valid in only one execution context. Therefore, the opaque data structure that a context handle refers to on a client implicitly includes a binding handle. Whenever an operation has an **in** or a non-null **in,out** context handle parameter, and the operation also has a first parameter that is of type **handle_t** or has the **handle** attribute, then the binding handle represented by the context handle and the binding handle represented by the first parameter must refer to the same execution context. Furthermore, when an operation has an **in** or a non-null **in,out** context handle parameter, any interface-wide binding mechanism — **implicit_handle** or **auto_handle** — is ignored for that operation. If an operation has more than one **in** context handle, all the respective binding handles must refer to the same remote execution context.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2 Interoperability Requirements on Stubs

Stub code shall conform to interoperability requirements in the following areas:

- operation number generation
- error handling when unmarshalling floating point data.

5.2.1 Operation Numbers

The RPC protocols use operation numbers to inform a server which operation of an interface to invoke. Stubs generate operation numbers consecutively, beginning with 0 (zero), in the order in which operations appear in the IDL source.

5.2.2 Error Handling During Floating-Point Unmarshalling

This subclause specifies how stubs handle errors that occur when unmarshalling floating-point data. The following list names a set of octet stream representations of floating point values for which stubs must generate errors:

MAX Some value greater than the largest value that can be represented in the receiver's floating-point representation.

MIN Some value less than the smallest value that can be represented in the receiver's floating-point representation.

NaN The logical equivalent to IEEE "not a number".

Minuszero The logical equivalent to IEEE –0.0.

+INF Positive infinity (in the format specified in the format label).

–INF Negative infinity.

Table 5-3 specifies stub behaviour for each of these conditions. The table indicates the value to be unmarshalled or a fault status code that must be returned by the caller (client).

Table 5-3 Floating Point Error Handling

Condition	Unmarshalled Value	Fault Status Code
MAX	undefined	rpc_s_fault_fp_overflow
MIN	0.0	None
Minuszero	0.0	None
+INF or NaN	undefined	rpc_s_fault_fp_error
–INF	undefined	rpc_s_fault_fp_error

When a floating-point error occurs on the server side, the server must return the appropriate fault PDU to the client to generate the fault status specified. The mapping of fault PDU values to fault status codes is specified in Annex E.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 6. Remote Procedure Call Model

This Section provides a high-level description of the Remote Procedure Call (RPC) model specified by this International Standard. Implementations must comply with the specified model in order to guarantee both application portability and interoperability between RPC peers.[3]

Note: For a description of the RPC model that provides guidelines for application program portability, see Section 2.

The RPC mechanism maps the local procedure call paradigm onto an environment where the calling procedure and the called procedure are distributed between different execution contexts that usually, but not necessarily, reside on physically separate computers that are linked by communications networks.

A procedure is defined as a closed sequence of instructions that is entered from, and returns control to, an external source. Data values may be passed in both directions along with the flow of control. A procedure call is the invocation of a procedure. A local procedure call and an RPC behave similarly; however, there are semantic differences due to several properties of RPCs:

- Server/client relationship (binding)

  While a local procedure call depends on a static relationship between the calling and the called procedure, the RPC paradigm requires a more dynamic behaviour. As with a local procedure call, the RPC establishes this relationship through binding between the calling procedure (client) and the called procedure (server). However, in the RPC case a binding usually depends on a communications link between the client and server RPC run-time systems. A client establishes a binding over a specific protocol sequence to a specific host system and endpoint.

- No assumption of shared memory

  Unlike a local procedure call, which commonly uses the call-by-reference passing mechanism for input/output parameters, RPCs with input/output parameters have copy-in, copy-out semantics due to the differing address spaces of calling and called procedures.

- Independent failure

  Beyond execution errors that arise from the procedure call itself, an RPC introduces additional failure cases due to execution on physically separate machines. Remoteness introduces issues such as remote system crash, communications links, naming and binding issues, security problems, and protocol incompatibilities.

- Security

  Executing procedure calls across physical machine boundaries has additional security implications. Client and server must establish a security context based on the underlying security protocols, and they require additional attributes for authorising access.

3. Implementations must comply with this specification regardless of the underlying transport protocol. Security protocols other than the one currently specified in this International Standard may also behave differently. Although these protocol specifics should be considered as part of the specification of the protocol machines, appropriate explanations are also included in this part of the specification for clarity and better readability.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1 Client/Server Execution Model

The RPC model makes a functional distinction between clients and servers. A client requests a service, and a server provides the service by making resources available to the remote client.

6.1.1 RPC Interface and RPC Object

Two entities partially determine the relationship between a client and a server instance: RPC interfaces and RPC objects. Both interfaces and objects are identified by UUIDs. (See Annex A for a UUID specification.)

6.1.1.1 RPC Interfaces

An RPC interface is the description of a set of remotely callable operations that are provided by a server. Interfaces are implemented by **managers**, which are sets of server routines that implement the interface operations. RPC offers an extensive set of facilities for defining, implementing and binding to interfaces. RPC explicitly imposes only a few restrictions on the behaviour of interface implementations. These include the following:

Execution Semantics Because RPC calls depend on network transports that provide varying guarantees of success, interface specifications include an indication of the effects of multiple invocations. Managers must be consistent with the specified semantics.

Version Numbering RPC provides a mechanism to specify interface versions and a protocol to select a compatible interface at bind time. Managers must provide the required version compatibility; that is, they are required to support the specified interface major version and the minor versions that are less than or equal to the minor version number of the interface advertised by the server.

An **interface identifier** is a UUID that uniquely identifies the RPC interface being called. Interface UUIDs are mandatory and are included in the interface specification in IDL (see Section 4).

6.1.1.2 RPC Objects

RPC objects are either server instances or other resources that are operated on and managed by RPC servers, such as devices, databases and queues. Servers here are the instances of services (applications) that are provided to RPC clients. Binding to RPC objects is facilitated by RPC, but object usage is optional and in the domain of application policies. Hence, RPC objects provide a means of object-oriented programming in the RPC environment, but allow applications to determine how these entities are actually being implemented. The object identifier is a UUID, called an **object UUID** that uniquely identifies the object on which the RPC is operating.

Object UUIDs for server instances and for resources cannot be intermixed. If multiple server instances are distinguished via object UUIDs (also called instance UUIDs), each binding operation only supports a single embedded object UUID. If the usage of multiple object UUIDs is required, these may be passed as explicit call arguments.

Servers may refer to multiple RPC objects, and RPC objects may be referenced by multiple servers; servers typically use different object UUIDs to refer to the same RPC object. RPC objects may be accessed by operations defined by one or a set of RPC interfaces.

To identify classes of RPC objects, these may also be tagged with **type UUIDs**. RPC has no predefined notion of an object or types of objects, but managers at the server may associate a type with an object. Type UUIDs are set to the nil UUID by default. Type UUIDs can only be assigned to RPC objects with non-nil UUIDs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.1.2 Interface Version Numbering

A client may bind to a server for a particular interface only if the client interface meets with the following conditions:

a) The client interface has the same UUID as the server interface.

b) The client interface has the same major version number as the server interface.

c) The client interface has a minor version number that is less than or equal to the server interface's minor version number.

6.1.2.1 Rules for Changing Version Numbers

From the version numbering rules, it can be seen that the minor version number is used to indicate that an upwardly compatible change has been made to the interface. The rules for changing version numbers are as follows:

a) The minor version number must be increased any time an upwardly compatible change or set of upwardly compatible changes is made to the interface definition.

b) The major version number must be increased any time any non-upwardly compatible change or set of changes is made to the interface definition.

c) If a change is made that requires a major version number increase, any upwardly compatible changes may be made at the same time; changing the minor version number is not required in this case, although it is permissible, and it is recommended that the minor version number be reset to 0 (zero).

d) The major version number can never be decreased.

e) The minor version number cannot be decreased without simultaneously increasing the major version number. These rules lead to the following guidelines for the use of version numbers:

— The initial values for the major and minor version numbers of an interface should be small. The values 1 and 0, yielding the version number 1.0, are typical.

— The increment used when increasing the major or minor version number is usually 1.

6.1.2.2 Definition of an Upwardly Compatible Change

The following are upwardly compatible changes, and may be made to an existing interface definition provided the minor or major version number is increased:

- Adding an operation to the interface, if and only if the operation is placed lexically after all the existing operations in the IDL source.
- Adding a type definition or constant, provided the new type definition or constant is used only by operations added at the same time, or later.

6.1.2.3 Non-upwardly Compatible Changes

Any change to an existing interface definition not listed in Subclause 6.1.2.2 is not upwardly compatible and requires an increase to the major version number.

6.1.3 Remote Procedure Calls

A specific remote operation, equivalent to a local function call in C, is instantiated by one RPC. The operation performed by an RPC is determined by the interface (identifier and version) and the operation number. Each instance of an RPC is uniquely identified by a distinct pair of session and call identifiers.[4]

A session is uniquely determined by the activity (connectionless protocol) or association (connection-oriented protocol). Sessions can be serially reused, but concurrent multiplexing of sessions is not supported.

---

4. "Session" in this context refers to an established client/server relation. This is expressed as an **activity** for the connectionless RPC protocol. The connection-oriented protocol defines this as **association**: a communications channel shared between a client's and a server's endpoint. Local policy governs the number and lifetime of sessions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Multiple session identifiers may correspond to a single client and server execution context pair, which is identified by the **cas_id** (connectionless protocol, obtained through the conversation manager handshake) or the **assoc_group_id** (connection-oriented protocol).

6.1.4 Nested RPCs

A called remote procedure can initiate another RPC. The second RPC is nested with the first RPC. Initial and nested RPCs are distinct according to the definition of an RPC; they are different RPC threads and operate on distinct sessions.

A specialised form of nested RPC involves a called remote procedure that makes an RPC to the execution context of the calling client application thread. Calling the original client's execution context requires that a server application thread is listening in that execution context. Also, the second remote procedure needs a server binding handle for the execution context of the calling client.

6.1.5 Execution Semantics

Execution semantics identify how many times a server-side procedure may be executed during a given client-side invocation. The guarantees provided by the RPC execution semantics are independent of the underlying communications environment. All invocations of remote procedures risk disruption due to communications failures. However, some procedures are more sensitive to such failures, and their impact depends partly on how reinvoking an operation transparently to the client affects its results.

The operation declarations of an RPC interface definition indicate the effect of multiple invocations on the outcome of the operations. The **at-most-once** execution semantic guarantees that operations are not executed multiple times.

The execution semantics for RPCs are summarised in Table 6-1.

Table 6-1 Execution Semantics

<table>
<tr><th>Semantics</th><th>Meaning</th></tr>
<tr><td>at-most-once</td><td>The operation must execute either once, partially, or not at all. This is the default execution semantics for remote procedure calls (also called non-idempotent).</td></tr>
<tr><td>idempotent</td><td>The operation can execute more than once. The manager routine must assure that executing more than once using the same input arguments produces identical outcomes without undesirable side effects.

An implementation of the RPC protocol machines may treat an idempotent call request as a non-idempotent call. This is a valid transformation.

RPC supports maybe semantics and broadcast semantics as special forms of idempotent operations.
<table>
<tr><th>Semantics</th><th>Meaning</th></tr>
<tr><td>maybe</td><td>The caller neither requires nor receives any response or fault indication for an operation, even though there is no guarantee that the operation completed. An operation with maybe semantics is implicitly idempotent and must lack output parameters.</td></tr>
<tr><td>broadcast</td><td>The operation is always broadcast to all host systems on the local network, rather than delivered to a specific server system. An operation with broadcast semantics is implicitly idempotent; broadcast semantics are supported only by connectionless protocols.</td></tr>
</table>
</td></tr>
</table>

With the RPC communications protocols, a **maybe** call lacks execution guarantees; an **idempotent** call, including **broadcast**, guarantees that the data for an RPC is received and processed zero or more times; and an **at-most-once** call guarantees that the call data is received and processed at most one time (may be executed partially or zero times). Both **idempotent** and **at-most-once** services guarantee that a sequence of

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

calls in a session are processed in the order of invocation by the client.

6.1.6 Context Handles

Server application code can store information it needs for a particular client, such as the state of previous RPCs the client made, as part of a client context. During a series of remote procedure calls, the client may need to refer to the client context maintained by a specific server instance. To provide a client with a means of referring to its client context, the client and server pass back and forth an RPC-specific parameter called a context handle. A **context handle** is a reference to the server instance and the client context of a particular client. A context handle ensures that subsequent RPCs from the client can reach the server instance that is maintaining context for the client (commonly known as "stateful" servers).

On completing the first procedure in a series, the server returns a context handle to the client. The context handle identifies the client context that the server uses for subsequent operations. The client stores the handle and can return it unchanged in subsequent calls to the same server. Using the handle, the server finds the context and provides it to the called remote procedure.

The server maintains the client context for a client until one of the following occurs:

- The client calls an operation that terminates use of the context.
- The server crashes.
- Communications are lost and the server provider invokes a context rundown procedure.

For a specification of the **context_handle** attribute, its usage, and its relation to binding handles, see Subclause 4.2.16.6 on page 230.

6.1.7 Threads

Each RPC occurs in the context of a thread. A **thread** is a single sequential flow of control with one point of execution at any instant. A thread created and managed by application code is an **application thread**.

RPC applications use application threads to issue both RPCs and RPC run-time calls. An RPC client contains one or more **client application threads**, each of which may perform one or more RPCs. (A client application thread may not make any RPC, or zero calls may be performed if a communications failure was detected.)

In addition, for executing called remote procedures, an RPC server uses one or more **call threads** that the RPC run-time system provides. When beginning to listen, the server application thread specifies the maximum number of concurrent calls it will execute. Single-threaded applications have a maximum of one call thread. The maximum number of call threads in multi-threaded applications depends on the design of the application and RPC implementation policy. The RPC run-time system creates the call threads in the server execution context.

An RPC extends across client and server execution contexts. Therefore, when a client application thread calls a remote procedure, it becomes part of a logical thread of execution known as an **RPC thread**. An RPC thread is a logical construct that encompasses the various phases of an RPC as it extends across actual threads of execution and the network. After making an RPC, the calling client application thread becomes part of the RPC thread. Usually, the RPC thread maintains execution control until the call returns.

The RPC thread of a successful RPC moves through the execution phases illustrated in Figure 6-1 on page 272.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

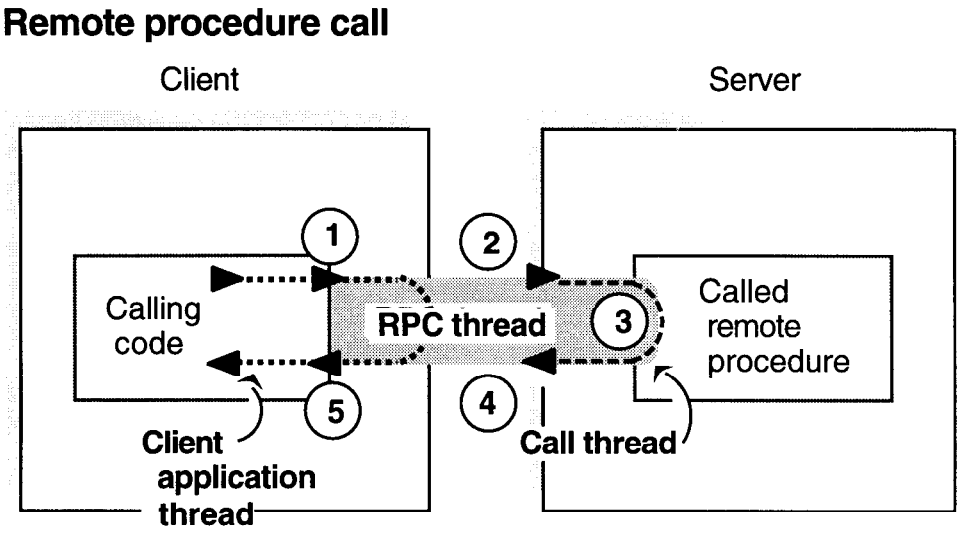


Figure 6-1 Execution Phases of an RPC Thread

The execution phases of an RPC thread, as shown in Figure 6-1, include the following:

a) The RPC thread begins in the client process, as a client application thread makes an RPC to its stub; at this point, the client thread becomes part of the RPC thread.

b) The RPC thread extends across the network to the server.

c) The RPC thread extends into a call thread, where the remote procedure executes.

   While a called remote procedure is executing, the call thread becomes part of the RPC thread. When the call finishes executing, the call thread ceases being part of the RPC thread.

d) The RPC thread then retracts across the network to the client.

e) When the RPC thread arrives at the calling client application thread, the RPC returns any call results and the client application thread ceases to be part of the RPC thread.

Figure 6-2 on page 273 shows a server executing remote procedures in its two call threads, while the server application thread listens.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

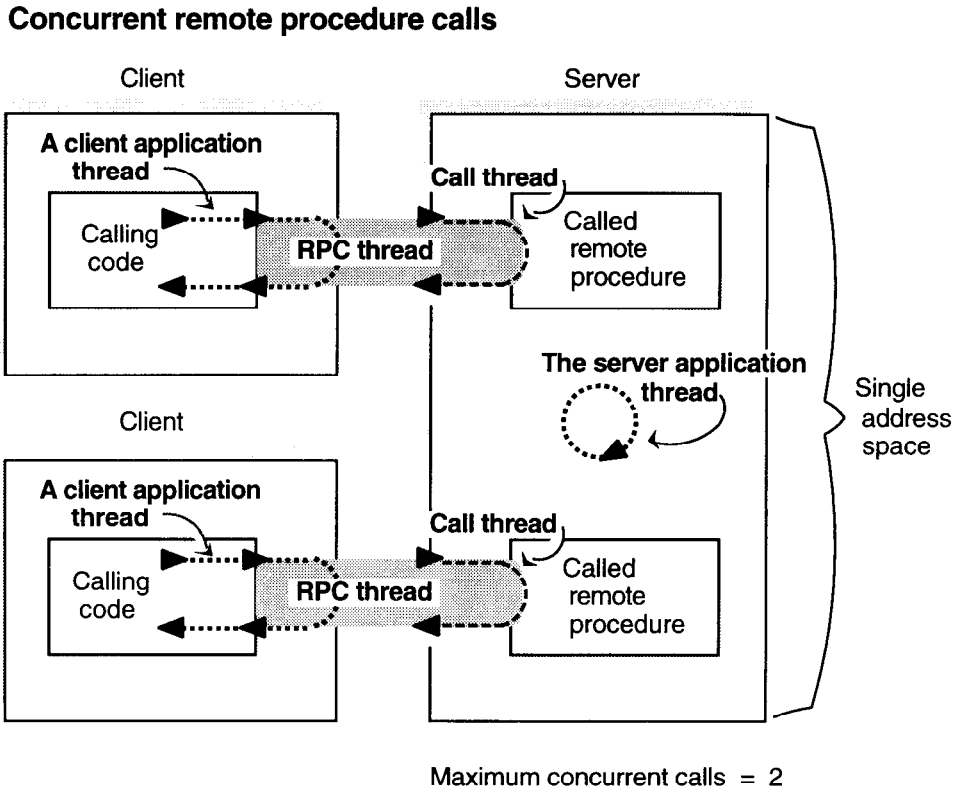


Figure 6-2 Concurrent Call Threads Executing in Shared Execution Context

Note: Although a remote procedure can be viewed logically as executing within the exclusive control of an RPC thread, some parallel activity may occur in both the client and server that is transparent to the application code.

An RPC server can concurrently execute as many RPCs as it has call threads. When a server is using all of its call threads, the server application thread may continue listening for incoming RPCs. While waiting for a call thread to become available, the RPC server run-time environment may queue incoming calls. Queuing incoming calls avoids RPCs failing during short-term congestion. This queue capability for incoming calls is implementation-dependent.

6.1.8 Cancels

A cancel is an asynchronous notification from a cancelling thread to a cancelled thread, generally used to cancel an operation in progress. The RPC architecture extends the semantics of cancels to incorporate RPCs.

In the absence of an RPC, both the thread initiating a cancel and the thread to be cancelled must belong to the same local execution context. In the presence of an RPC, the desired semantic is that the system should behave as if the remote procedure were local and part of the cancelled thread's execution context. That is, if a thread has called a remote procedure, is waiting for the remote procedure to complete, and is cancelled, its RPC run-time system will handle the cancel and forward it to the called procedure's RPC run-time system, where it will locally cancel the thread running the called procedure.

RPC forces the convention that the ability to cancel asynchronously must be lexically scoped (in the same lexical unit, such as a function or procedure). Therefore, at the completion of an RPC, the RPC run-time system will always restore the asynchronous delivery state prior to the call, regardless of any unbalanced asynchronous cancellability that may exist within the RPC. This behaviour may be different from the local case, where unbalanced asynchronous cancelability may not be detected. (For further information on the semantics of threads and cancels, see IEEE P1003.4a.)

Well-behaved programs must also observe the convention that general cancelability must be lexically scoped. If the caller is within a general cancelability disabled scope at the time an RPC is called, RPC will never see the cancel; it will only become visible after the RPC completes and the caller ends the general

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

cancellability disabled scope.

Well-behaved remote procedures, as well as the RPC system, do not pass their thread identity to any other (user) threads, and therefore cannot be locally cancelled. There is one exception to this: if the RPC run-time system ascertains that communications are lost, it cancels the called procedure to initiate its orderly termination. Therefore, any remote procedure must still protect its invariants with a suitable general and asynchronous cancellability scope. RPC must provide a means of specifying that a remote procedure begins (and ends) its execution in a disabled scope for either general or asynchronous cancellability in order to avoid a race condition between the beginning of the procedure and establishing the cancellability scopes within the procedure.

Cancels operate on the RPC thread exactly as they would on a local thread, except for an application-specified cancel time-out period. A **cancel time-out period** is an optional value that limits the amount of time the cancelled RPC thread has before it releases control. This timer allows the caller to guarantee that it can reclaim its resources and continue execution within a bounded time. The timer may be set to an "infinite" value, in which case the caller will wait indefinitely until the called procedure returns (usually with a cancelled exception) or communications are lost. The timer may be set on a per RPC basis.

During an RPC, if its thread is cancelled and the cancel time-out period expires before the call returns, the calling thread regains control and the call is orphaned at the server. An orphaned call may continue to execute in the call thread. However, the call thread is no longer part of the RPC thread, and the orphaned call is unable to return results to the client; the caller does not know whether or not the called routine has terminated yet, how it may have terminated, or even if it executed.

While executing as part of an RPC thread, a call thread can be cancelled only by a client application thread. The local cancel semantics can be guaranteed for all RPCs that do not fail due to server or communication errors. That is, cancels can be transferred remotely to or from the called procedures. In the case where an RPC fails due to either server or communication failures, it is indeterminate whether cancels were preserved, just as it is indeterminate whether the procedure executed zero or one time.

The RPC architecture specifies neither what causes a cancel, nor what an application does when cancelled. This is application-specific. Nor does the architecture place any semantics on the cancel; again the application must decide what it means.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2 Binding, Addressing and Name Services

The following subclauses cover binding, endpoint addresses and name services.

6.2.1 Binding

Binding expresses the relationship between a client and a server. Binding includes information that associates the client's invocation of an RPC with the server's implementation (that is, the manager routines) of the call. The binding information identifying a server to a client is called **server binding information**. Binding information identifying a client to a server is called the **client binding information**.

To make a specific instance of locally maintained binding information available to a given server or client, the RPC run-time system creates a local reference, called the **binding handle**. Servers and clients use binding handles to refer to binding information in RPC run-time calls or remote procedure calls.

Binding information includes the following components:

Protocol Sequence

The protocol sequence is a valid combination of communications protocols. Each Protocol sequence typically includes a network protocol, a transport protocol, and an RPC protocol.

An RPC server specifies to the RPC run-time system the set of protocol sequences to use when listening for incoming calls.

Network Address Information

The network address provides the complete transport service address information of the remote entity. Typically, for commonly used network protocol stacks such as Internet, the targetted entity is determined by nodes or the host system. In these instances, the network address information includes:

- A **node address**, which identifies a specific host on a network. The format of the address depends on the network protocol determined in the protocol sequence.
- An **endpoint**, which specifies the address of a specific server instance. The format of the endpoint depends on the network protocol determined in the protocol sequence. Endpoints are unique for each protocol sequence and for each server listening on a given network address.

Transfer Syntax

The server must support a transfer syntax that matches one used by the client. For multi-canonical transfer syntaxes such as NDR, a given sender's data representation format must be understood by the receiver.

RPC Protocol Version Numbers

The client and server RPC run-time systems must use compatible versions of the RPC protocol specified by the client in the protocol sequence. The major version number of the RPC protocol used by the server must equal the specified major version number. The minor version number of the RPC protocol used by the server must be greater than or equal to the client's specified minor version number.

Object UUID

The object UUID associated with the binding information is optional.

RPC run-time system creates one or more server binding handles for each protocol sequence. Each server binding handle refers to binding information for a single potential binding. A server obtains a complete list of its binding handles from its RPC run-time system.

A client obtains a single binding handle or a set of binding handles from its RPC run-time system. It selects one binding handle for invoking one or a sequence of RPCs to a given server. Server binding information for each server binding handle on a client contains binding information for one potential binding.

If the network address in the server binding information on a client refers to a "host-addressable" network service, it may be partial, lacking an endpoint. A partially bound binding handle corresponds to a system, but not to a particular server instance. When invoking a remote procedure call using a partially bound binding handle, a client gets an endpoint either from the interface specification or from an endpoint map on the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

server's system. Adding the endpoint to the server binding information results in a fully bound binding handle.

6.2.2 Endpoints and the Endpoint Mapper

An **endpoint** is the address of a specific server instance on a host system. Two types of endpoints exist: well-known endpoints and dynamic endpoints.

- A **well-known endpoint** is a preassigned, stable address for a particular server instance. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol.

  Well-known endpoints can be declared for an RPC interface (in the interface declaration) or for a server instance.

- A **dynamic endpoint** is an endpoint that is requested and assigned at run time.

The **endpoint mapper** is an RPC service that manages dynamic endpoints. The remainder of this subclause specifies the services offered by an endpoint mapper, and discusses how the RPC run-time system uses those services.

The endpoint mapper service may only be applicable to systems that provide "host-addressable" transport services. The notions of **endpoints** and **well-known endpoints** are derived from the Internet Protocol Suite, but may be applicable to other network protocol stacks as well. In order to provide for application portability, it is mandatory, when dynamic endpoints are used, that RPC implementations on systems with these types of transport services comply with this specification.

An endpoint mapper may be used to help resolve the address of a server. This is typically used with network addresses that have a small range of values for the local endpoint address (for example, an IP port) and/or by servers that want to dynamically define an endpoint address. Typically, in such cases a server exports its node address to the name service. The endpoint mapper's endpoint address is well known. The server also registers its interfaces, interface version and object UUIDs with its local endpoint mapper, along with a dynamically determined local endpoint address.

An RPC client wishing to use the server will (typically) query the name service to determine the address, using one of the RPC name service APIs. The address returned includes a value that signifies the endpoint mapper endpoint, which is a well-known endpoint (see Annex H).

An erroneous or malicious endpoint mapper implementation can cause denial of service, but otherwise does not affect the security of the system.

6.2.2.1 Client Operation

The use of the endpoint mapping service is transparent to client name service operations.

At the client stub to RPC run-time interface, every RPC specifies a primitive binding handle that includes the server address. If the system-specific endpoint address specified is one of the well-known endpoint addresses for the endpoint mapping service, and the interface specified is not the endpoint mapping service interface[5], then the endpoint mapping service on the desired target system is requested to resolve the partially bound server binding handle into a fully bound server binding handle.

The client run-time system of a connection-oriented RPC issues a call to the endpoint mapping service on the desired target system prior to the originating call. When the call successfully completes, the effective endpoint for the binding handle is set to the dynamically determined value returned by the endpoint mapping service. This endpoint is then used to make the actual call requested.

The client run-time system of a connectionless RPC issues the first request of the originating call with a partially bound server binding handle. The endpoint mapping service resolves this partially bound handle into a fully bound server binding handle and redirects the call. The server then returns the dynamically determined value directly to the client for use in subsequent messages.

If the request for resolving the partially bound server binding handle into a fully bound server binding handle fails, then the originating RPC fails with an error status.

---

5. This terminates the recursion.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.2.2 Server Operation

The use of an endpoint mapping service is transparent to the call and the server RPC run-time system with one exception: with a dynamically assigned port, when the server exports binding information to a name service, the export operations must export a value that signifies the endpoint mapper service rather than the dynamically assigned port.

6.2.3 NSI Interface

The RPC architecture requires a means to allow clients to discover appropriate servers. This specification defines the use of a distributed name service to store information about servers, service groups and configuration profiles. A candidate name service must be able to store all the object attributes specified here. Multiple name services may satisfy this requirement, but a client and server can only bind successfully through a name service if they share use of some common information base.

Each name service object entry consists of a number of attributes. This RPC specification requires a small number of different name service object attributes. Additional name service object attributes provided by some name services may be ignored by RPC. Attributes have the following characteristics:

- They are either single-valued or multi-valued (set-valued).
- A single value or member of a set must support at least 4000 octets.
- There is no architectural limit to the number of elements in a set.

If a redundant value is inserted in the set, a new entry is not made. If a non-existent value is removed from a set, no error is generated. The order of elements in a set is not defined, and any order observed is neither significant nor deterministic; that is, implementations may vary, but applications must not make any assumptions on the ordering.

Different name services may have different syntaxes to represent object names; their object name syntax is not specified in the RPC specification. The RPC operations that use object names require the different syntaxes to be explicitly distinguished to avoid ambiguity and to allow the implementations to interpret the name values properly. Since different name services also may have different conventions for naming attributes, and since the names of the attributes are not directly user visible through the RPC services, for each different name service there is a mapping from the defined class names to name service-specific names.

6.2.3.1 Common Declarations

The following declarations define the name service data types required for RPC:

```
typedef char    class_name_t[31];       /* ISO_LATIN_1 Attribute class name */

typedef struct {
                byte        major;
                byte        minor;
          } class_version_t;

/* Opaque octet string */
typedef struct {
                u_int16           count;          /* store little-endian */
                [ptr, size_is(count)]  byte  *value;
          } octet_string_t;

/* One layer in a protocol tower */
typedef struct {
                octet_string_t   protocol_id;
                octet_string_t   address;
          } prot_and_addr_t;

/* A protocol tower */

typedef struct {
                u_int16          count;          /* store little-endian */
                [ptr, size_is(count)]  prot_and_addr_t  *floors;
          } protocol_tower_t;

/*
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
 * Name service names are stored as canonical string names
 * according to the rules for the relevant name service.
 */
typedef byte    canonical_string_name_t[];        /* Using ASCII encoding */

/* An element within a profile.
 * There may be multiple set members for the same interface.
 * The UUID NIL with versions 0 indicates the default profile,
 * i.e. linkage to a parent profile.
 */
typedef struct {
                uuid_t          if_uuid;        /* store little-endian */
                u_int16         if_vers_major;  /* store little-endian */
                u_int16         if_vers_minor;  /* store little-endian */
                u_int8          priority;       /* legal values are 0
                                                   (highest) through 7 */
                u_int8          annot_size;     /* annotation size */
                u_int16         member_size;    /* member size, store
                                                   little-endian */
                [size_is(annot_size)]   byte annotation[]; /* ASCII encoding*/
                [size_is(member_size)]  canonical_string_name_t member;
          } profile_element_t;
```

Note: The **protocol_tower_t** data type is encoded using special rules defined in Annex L. It is then cast into a **byte**[] type for use in the **tower_octet_string** field of the **twr_t** and ***twr_p_t** types, as defined in Annex N, and used in the end-point mapper interface.

6.2.3.2 Protocol Towers

In order to communicate, the RPC client and server must agree both upon the protocols that both will employ, and upon the operational parameters of these protocols. In addition, client and server must possess address information that indicates to each layer of protocol where to deliver data.

A **protocol tower** (encoded by the **protocol_tower_t** data type) is a protocol sequence along with its related address and protocol-specific information. A protocol sequence is an ordered list of protocol identifiers. **Protocol identifiers** are octet strings, each representing a distinct protocol at some layer.

Addressing and other protocol specific information is affiliated with each protocol identifier in a protocol tower. The addressing information indicates the access point through which this layer provides service to the next higher layer protocol in the sequence. Other protocol-specific information may be included in this field. The interpretation of this address and other information is protocol-dependent. Typically, a protocol sequence will extend from the network layer to the application layer.

An RPC client and server must have at least one common protocol tower where the protocol identifiers (the left-hand sides) match. Otherwise they do not share a common stack of protocols and cannot communicate.

Table 6-2 on page 279 shows the generic structure for a protocol tower. It shows three layers to illustrate the relationships among adjacent layers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 6-2 Protocol Tower Structure

Protocol Identifier for	Identifier Value	Related Information
Layer **i**+1 protocol identifier	Layer **i**+1 value	Layer **i**+1 parameters and address data selecting layer **i**+2 protocol.
Layer **i** protocol identifier	**i** value	Layer **i** parameters and address data selecting layer **i**+1 protocol.
Layer **i**–1 protocol identifier	**i**–1 value	Layer **i**–1 parameters and address data selecting layer **i** protocol.

6.2.3.3 The server_name Object Attributes

The **server_name** attributes of a single name service entry describe a single RPC server (that is, instance) and its protocol and addressing information. Any name service object class may contain **server_name** attributes if not otherwise prohibited by the class.

The class **RPC_Entry** may be used if no other class is applicable.

The hierarchy of protocols and addresses is expressed in terms of a **protocol_tower** data type. The **server_name** object attributes are defined in Table 6-3 on page 280.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table 6-3 The server_name Object Attributes

Attribute Name	Single or Set Valued	Data Type	Description
CDS_Class	Single	**class_name_t**	An existing class, or **RPC_Entry** if created by RPC.
CDS_ClassVersion	Single	**class_version_t**	An existing class version of the class definition or, 1.0 if created by RPC.
RPC_ClassVersion	Single	**class_version_t**	Version 1.0; may already exist.
RPC_ObjectUUIDs	Set	**uuid_t, little-endian order**	Optional UUIDs of the referenced server objects.
CDS_Towers	Set	**protocol_tower_t**	The set of protocol towers for this server.

The **CDS_Towers** attribute must encode both the RPC-specific protocol layers, and the underlying network, transport, session and presentation layers, as applicable. The RPC-specific layers are "on top" (lowest array subscripts) and are specified in Table 6-4.

Table 6-4 RPC-specific Protocol Tower Layers

Protocol Identifier for	Identifier Format	Related Information	Comments
Interface, major version	**UUID_type_identifier**	The minor version, **u_int16**, little-endian order.	Value derived from encoding algorithm (see Annex I).
Transfer Syntax, major version	**UUID_type_identifier**	The minor version, **u_int16**, little-endian order.	Value derived from encoding algorithm (see Annex I).
RPC Protocol and major version	**u_int8**	The minor version **u_int16**, little-endian order.	See Annex I for identifier values, according to RPC protocol and major version number.

The encoding of the protocol identifier for a particular interface, or for a particular transfer syntax is specified in Annex I.

The other layers depend on the particular environment. Annex I defines protocol identifier values for common environments. Table 6-5 on page 281 shows an example of a complete tower for a TCP/IP-based protocol.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 6-5 Example Protocol Tower

Protocol Identifier for	Identifier Value	Related Information	Comments
Interface, major version	**UUID_type_identifier**	Minor version	Value derived from encoding algorithm (see Annex I).
NDR V1.1 Transfer Syntax	**UUID_type_identifier**	—	Value derived from encoding algorithm (see Annex I).
RPC CO protocol, major version	0b hexadecimal	Minor version	—
DOD TCP	07 hexadecimal	Port	Port address is 16-bit unsigned integer, big-endian order.
DOD IP	09 hexadecimal	Host address	Host address is 4 octets, big-endian order.

6.2.3.4 The group Object Attributes

A name service **group** attribute refers to a management defined group of equivalent servers. Any name service object class may contain a group attribute if not otherwise prohibited by the class. The class **RPC_Entry** may be used if no other class is applicable.

Each element of the set **RPC_Group** is of the data type **canonical_string_name_t** and represents the name of another name service object containing either a name service **server_name** attribute or another name service group attribute.

The group object attributes are defined in Table 6-6 on page 282.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 6-6 Service Group Object Attributes

Attribute Name	Single or Set Valued	Data Type	Description
CDS_Class	Single	**class_name_t**	An existing class, or **RPC_Entry** if created by RPC.
CDS_ClassVersion	Single	**class_version_t**	An existing class version of the class definition or 1.0 if created by RPC.
RPC_ClassVersion	Single	**class_version_t**	Version 1.0; may already exist.
RPC_Group	Set	**canonical_string_name_t**	The set of server object names or service group names for this **service_group**.

6.2.3.5 The profile Object Attributes

A name service **profile** attribute refers to a principal or host's desired server profile. Any name service object class may contain a profile attribute if not otherwise prohibited by the class. The class **RPC_Entry** may be used if no other class is applicable.

Each element of the set attribute **RPC_Profile** is of the data type **profile_element_t** and represents an ordered list of providers for a particular interface (UUID). A profile with the nil interface UUID indicates the default profile to use if no matching interface is found. Each profile element contains an ordered list of the names of name service objects containing any combination of **server_name**, group and/or profile attributes.

The **profile** object attributes are defined in Table 6-7 on page 283.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 6-7 Configuration Profile Object Attributes

Attribute Name	Single or Set Valued	Data Type	Description
CDS_Class	Single	**class_name_t**	An existing class, or **RPC_Entry** if created by RPC.
CDS_ClassVersion	Single	**class_version_t**	An existing class version of the class definition, or 1.0 if created by RPC.
RPC_ClassVersion	Single	**class_version_t**	Version 1.0; may already exist.
RPC_Profile	Set	**profile_element_t**	The set of providers comprising the configuration profiles.

6.2.3.6 Encoding

The encoding of the name service objects may be viewed from three perspectives:

- From the perspective of the RPC API to the name service operations and data types, the representation is as defined in Subclause 6.2.3.1 on page 277 to Subclause 6.2.3.5 on page 282.
- From the perspective of the name service communications, the encoding is defined by the network encoding rules of the name service.
- From the perspective of the name service storage elements, the encoding is defined internally to the name service operation.

6.2.3.7 Name Service Class Values

A name service entry storing RPC attributes uses the class value **RPC_Entry** if no other class applies.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.3 Error Handling Model

The RPC service detects various classes of unusual or exceptional terminations of an RPC. These failure cases are either originated in the server application and manager routines, detected and raised in the server run-time system, or are communications failures detected locally in the client RPC run-time system.

Fault status conditions (**fault** PDU) always indicate error conditions that are generated in the manager routines or server application. The server protocol machine does not process fault status codes.

Reject status conditions (**reject** PDU in connectionless protocol, **fault** PDU in connection-oriented protocol) usually originate in the protocol machines or the underlying resources (communications, systems) and may require additional processing such as clean up of resources at the server protocol machine. The following set of reject messages indicate that a failed call has not been executed at the RPC server:

- unknown_interface
- unsupported_type
- manager_not_entered
- op_range_error
- who_are_you_failed
- wrong_boot_time.

Unless the protocol machine can detect the execution state by some other means (connection-oriented protocol), none of the other reject and fault conditions can determine whether a call has already been partially executed.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Section 7. RPC Service Definition

This Section specifies the basic operations performed by a Remote Procedure Call (RPC). These operations, which represent the interaction between the service user and the service provider, are specified as service primitives.

The service users are represented in the client stub and client application code and in the server stub, server manager routines and server application code, which provide the appropriate parameter values and process the results.

The service provider is represented in the RPC run-time system and is specified in the protocol machines, which generate and receive the events driven by the service primitives.

7.1 Call Representation Data Structure

The **call representation** data structure is an input parameter to all service primitives described here. Some information is dynamically generated during the sequence of common service operations that comprise an RPC.

The call representation data structure contains all of the control information about an individual RPC. This includes location and interface information. An interface specification and an operation number define the interface information. Dynamic information includes the transfer syntax in which the RPC arguments are marshalled, and cancel state information.

An RPC protocol machine may have specific information particular to an RPC. To facilitate this, the call representation data structure is composed of a common part and a protocol-specific part. A **call handle** is provided to the RPC stubs as an opaque pointer to the the call representation data structure.

7.2 Service Primitives

The tables in this Section, which specify the parameters that are present in each service primitive, use the following notation, as described in the ISO TR 8509 standard:

M The parameter is mandatory. It will always be present in the service primitive.

U The parameter is a user option. It need not be provided by the user in the particular instance of the service primitive.

C The parameter is conditional. It will always be present in that indication-type primitive if it was present in the corresponding request-type primitive.

(=) When the parameter is present in a particular instance of that indication-type primitive, it takes the same value it had in the corresponding request-type primitive.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.1 Invoke

The **Invoke** service primitive is used to invoke an RPC. The **Invoke** service primitive is service user-initiated.

Table 7-1 lists the parameters of the **Invoke** service primitive.

Table 7-1 Invoke Parameters

Parameter Name	Request	Indication
Call_Handle	M	M
Call_Args	U	C(=)
Call_Status	—	M

The permitted parameter values are as follows:

Call_Handle — The call handle that uniquely identifies this RPC.

Call_Args — The marshalled call arguments according to the parameters specified in the interface operation (IDL), if any.

Call_Status — A value indicating the status of the operation. For a summary of possible condition values that this operation can return, refer to Annex E.

The events and triggering conditions generated by this service primitive are:

START_CALL, TRANSMIT_REQ

The client user issues an RPC and generates the event START_CALL. The conditional flag TRANSMIT_REQ indicates that there is data in the send queue (request-type primitive).

RECEIVE_PDU[REQUEST_PDU]

The server provider generates the event RECEIVE_PDU (conditionally REQUEST_PDU) upon receiving a REQUEST PDU (indication-type primitive).

RCV_LAST_IN_FRAG

The server provider promotes the requested operation (including input data, if any) to the server user (server stub and manager routine) for execution.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.2 Result

The **Result** service primitive is used to return the output and input/output parameters at the end of a normal execution of the invoked RPC. The **Result** service primitive is server-user (server manager routine) initiated.

Table 7-2 lists the parameters of the **Result** service primitive.

Table 7-2 Result Parameters

Parameter Name	Response	Confirmation
Call_Handle	M	M
Result_Args	U	C(=)
Call_Status	—	M

The permitted parameter values are as follows:

Call_Handle — The call handle that uniquely identifies this RPC.

Result_Args — The marshalled output and input/output arguments according to the parameters specified in the interface operation (IDL), if any.

Call_Status — A value indicating the status of the operation. For a summary of possible condition values that this operation can return, refer to Annex E.

The events and triggering conditions generated by the **Result** service primitive are as follows:

PROC_RESPONSE, TRANSMIT_RESP
The server user processed this RPC request and generates the event PROC_RESPONSE. The conditional flag TRANSMIT_RESP indicates data in the send queue (response-type primitive).

RECEIVE_PDU[RESPONSE_PDU]
The client provider generates the event RECEIVE_PDU (conditionally RESPONSE_PDU) upon receiving a **response** PDU (confirmation-type primitive).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.3 Cancel

The **Cancel** service primitive is used to cancel an outstanding RPC. This operation forwards a client cancel request to the server application thread. If the server application thread does not return within a caller-specified time, the RPC will fail. Most of this processing is specific to the associated protocol machines.

Table 7-3 lists the parameters of the **Cancel** service primitive.

Table 7-3 Cancel Parameters

Parameter Name	Request	Indication
Call_Handle	M	M

The permitted parameter value is as follows:

Call_Handle — The call handle that uniquely identifies this RPC. The cancel-related data values and flags are modified appropriately.

The events and actions generated by the Cancel service primitive are:

CLIENT_CANCEL — The client user has issued a request to terminate a call in progress. The event CLIENT_CANCEL is generated.

RCV_CAN_PDU — The server provider receives a **cancel** PDU and generates the event RCV_CAN_PDU (RECEIVE_PDU, conditionally CANCEL_PDU).

CANCEL_NOTIFY_APP
The server provider notifies the server user about a pending cancel request.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.4 Error

The **Error** service primitive may be used by the server manager routine to indicate an error in response to a previous **Invoke** indication. The **Error** service primitive is server-user (server manager routine) initiated.

Table 7-4 lists the parameters of the **Error** service primitive.

Table 7-4 Error Parameters

Parameter Name	Response	Confirmation
Call_Handle	M	M
Call_Error_Value	M	M(=)

The permitted parameter values are:

Call_Handle The call handle that uniquely identifies this RPC.

Call_Error_Value The marshalled error information.

The events generated by the **Error** service primitive are as follows:

PROCESSING_FAULT
The server user detected a fault during execution of the requested operation and raises the event PROCESSING_FAULT.

RCV_FAULT The client provider generates the event RCV_FAULT upon receiving a **fault** PDU with fault status code.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.5 Reject

The **Reject** service primitive indicates that there is a problem with the underlying communications or the RPC protocol machines. The reject reason (parameter, Call_Reject_Reason) can indicate the state of a particular RPC and therefore may be evaluated to determine whether the call has already been executed at the server. The **Reject** service primitive is typically server provider-initiated. Some reject reasons such as Op_Range_Error are detected at the server user and are server user-initiated.

Table 7-5 lists the parameters of the **Reject** service primitive.

Table 7-5 Reject Parameters

Parameter Name	Response	Confirmation
Call_Handle	M	M
Call_Reject_Reason	M	M(=)

The permitted parameter values are:

Call_Handle — The call handle that uniquely identifies this RPC.

Call_Reject_Reason — The marshalled reject information. For a summary of which reject status codes are reported, refer to Subclause 12.4.4 on page 467.

The events generated by the Reject service primitive are as follows:

PROCESSING_FDNE — The server provider or server user rejects the execution of the requested operation (FaultDidNotExecute) and raises the event PROCESSING_FDNE.

RCV_FAULT — The client provider generates the event RCV_FAULT upon receiving either a **reject** PDU or a **fault** PDU with a reject status code.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 8. Statechart Specification Language Semantics

The protocol machines included in this International Standard are specified using the modelling technique proposed by David Harel and implemented by the software engineering tool Statemate (see subclause 1.3. The behavioural model of protocol machines is graphically expressed in statecharts and supported by a specific modelling language.

Statemate provides a complete software engineering tool including analyser and simulation, International Standard.r and prototyper. The definitions provided in this International Standard reflect only a subset of the Statemate semantics that have actually been used for RPC protocol machine specifications.

The logical semantics of **statecharts** are based on classical state transition diagrams using the specification technique of finite state machines. Statecharts introduce a number of significant extensions to overcome a major drawback of traditional finite state machines, which is their inherently flat and sequential nature. The complex behaviour of reactive systems such as protocol machines can be better expressed with statecharts. The most significant extensions are as follows:

- hierarchy of states
- concurrency between substates
- generation of internal events and broadcast of these within the entire chart.

The RPC protocol specification assumes knowledge of the semantics of traditional finite state machines. This Section only describes the statechart extensions.

8.1 The Elements of Statecharts

Similar to state transition diagrams, statecharts describe the following elements:

States States are the static elements in statecharts. The activation and deactivation of states is externally controlled. States may idle, perform actions or invoke operations, called **activities**. Actions that do not themselves cause a change in state are called **static reactions**. These may be performed within states when a trigger is sensed or they may be performed as a state is entered or exited.

States are represented graphically as rounded rectangles.

Triggers Triggers are the dynamic elements in statecharts. Triggers cause state transitions or static reactions. Events, conditions or a combination of both can be triggers.

Triggers represented graphically as directed graphs (arrows), connecting two states.

Events Events signify a precise instant in time; they are edge-sensitive, comparable to signals and interrupts. Events can be generated externally to the statechart (primitive events) or internally. Sources for internally generated events are actions, timeouts or sensors for detecting the status of states, activities, conditions and data items. Events may also be a compound set of other events and conditions.

Events are represented as a alphanumeric labels on transition lines or in the **On** field in state definitions.

Conditions Conditions are boolean expressions, valued TRUE or FALSE, that signify a time span during which the condition holds. Conditions can be edge or level-sensitive. Conditions may be primitive elements or compound elements that express a set of boolean operations such as AND and OR.

Conditions are represented as alphanumeric labels on transition lines,

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

enclosed in brackets in the form **event[condition]**, or in the **Trigger** field in state definitions.

Actions Actions are instantaneous operations and are performed as a result of some trigger. Changing the value of a condition or data item and invocations of activities are examples of actions. An action can also be a sequence of actions that occur simultaneously regardless of the sequence of their appearance. Logical conditional actions may be expressed based on conditions or on the occurrence of events.

Actions are represented as alphanumeric labels on the transition lines, separated by a slash in the form **event[condition]/action**, or in the **Action** field in state definitions. When an action is a sequence of actions, the actions that make it up are separated by semi-colons.

Activities Activities are operations that are performed in a non-zero amount of time. Activities can be controlled (started or stopped) through actions, and their status can be monitored.

Activities are not represented graphically in statecharts. They may appear as part of other element definitions.

Data Items Data items express values of primitive data types such as integer, real or character string. Their values can be changed through actions and evaluated in conditions.

Data items are not represented graphically in statecharts. They may appear as part of other element definitions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.2 State Hierarchies

Statecharts allow states to be nested to an arbitrarily deep level. Substates are represented graphically as states (rounded rectangles) within the parent state. Parent states are the superordinates or higher-level states. This hierarchical decomposition can be viewed as either clustering of logical groups of states (the bottom-up approach) or as refinement (the top-down approach).

If one substate is active, all its ancestor states are also active at the same time; and if a state containing substates (not a basic state) becomes active, one of the substates will also become active. A substate can transition out to any higher or lower-level state, including its immediate parent state.

Leaf states in this hierarchical structure are **basic** states, while all other states, which contain at least one descendent, are **non-basic** states. Any valid state transition must always affect a lowest level basic state or a set of basic states. (For the case of AND decomposition, see Subclause 8.3.) The source and target sets of a transition are always defined in terms of basic states.

8.3 Concurrency

Conventional state transition diagrams are purely sequential, because only one state can be active in the system at any given time. Permissible states can thus be modelled as a logical exclusive OR (XOR) of the possible states, or by regarding flow control as synchronous.

A state in the hierarchy of statecharts may contain an XOR decomposition of its substates, but statecharts also permit orthogonality. This is the dual of the XOR decomposition; in essence it is an AND decomposition. Such orthogonality introduces the notion of concurrency or asynchronous control.

AND states are multiple states that are active during the same time interval. When transitioning into one of the states in an AND decomposition, all other states become active simultaneously. Analogously, if a transition exits out of one of these orthogonal AND states, all other orthogonal states are deactivated as well. Hence, the concurrent states in an AND decomposition are always synchronously triggered.

As long as the concurrent states are active, substates in these states behave independently, asynchronously. Substate transitions do not affect the other active AND states in the system. However, synchronisation between these can be triggered through events; for example, they can be internally generated in one of the other AND states.

Orthogonal AND states are represented graphically as rounded rectangles (parent state) divided by dashed lines.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.4 Graphical Expressions

The following subclauses describe several additional graphic elements that support the semantics outlined before and provide for de-cluttering of statecharts. They are mainly introduced to improve the readability of statecharts.

8.4.1 Default Entrances

To prevent non-determinism for transitions into substates and orthogonal AND states, a default transition line must be applied to all these sets. These default entries determine which states become active initially unless some other directed transition is valid.

Default entries are represented graphically as a arrows emanating from dots. Default transitions are usually represented without transition labels.

8.4.2 Conditional Connectors

Condition connectors are syntactical graphic elements that are used to economise on arrows and declutter the chart. They are represented graphically as circles containing the letter C. Events, conditions and actions are labelled accordingly on inbound and outbound transition arrows. If distinct conditions apply to the outbound transition arrows, they must be exclusive in order to prevent nondeterminism.

8.4.3 Terminal Connectors

Terminal connectors are syntactical graphic elements that are used to express the termination of a statechart. They are drawn as circles containing the letter T. These T-connectors are considered as a final state; in particular, they have no exits. Upon entering this connector the statechart becomes deactivated.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.5 Semantics that Require Special Consideration

In order to determine the exact behaviour of the extended semantics applied to statecharts, the precise rules and dynamic control characteristics have to be defined. In most cases, knowledge of the functional behaviour of traditional finite state machines will be sufficient to read and understand statecharts, but, in particular, the concurrency model and state hierarchies require familiarity with the concepts described in the following subclauses:

8.5.1 Implicit Exits and Entrances (Scope of Transitions)

Transition arrows can be drawn between any two states in the system, including a loop back to the same state. This subclause defines how a transition — expressed by an arrow or a set of arrows connected by a conditional connector — is defined and performed. In general, a transition is to be considered as a **compound transition**, evaluating all attached events and conditions, and performing the entire set of effected actions.

In taking a transition from a source to a target, the transition will, in general, pass through different levels of the state hierarchy. Hence, the question arises as to which non-basic states are exited and entered in the process of taking a transition. This is especially important due to actions that may be called for when exiting and entering states.

To analyse this behaviour, the notion of the **scope** of a transition is introduced. The scope of a transition is the lowest XOR state in the hierarchy of states that is a proper common ancestor of all the sources and targets of a transition, including non-basic states that are explicit sources or targets of transition arrows appearing in the transition.

When a transition is taken, all proper descendants of the scope in which the system resides at the beginning of the step are exited, and all proper descendants of that scope in which the system will reside as a result of executing the transition are entered. Thus, the scope is the lowest state in which the system stays, without exiting and re-entering, when taking the transition.

8.5.2 Conflicting Transitions

Conflicting transitions are those that can lead the system into distinct states. Conflicting transitions are detected if there is a common state in their source sets; therefore, concurrent transitions across orthogonal states do not conflict. Multiple transitions that can occur at the same time step in a given scope have the same priority. Priority is given to the transition with the higher scope.

Conflicting states at the same priority must resolve non-determinism to be legal. Non-determinism occurs in cases where several different transitions are enabled at the same step (for example, a state with multiple outbound transition lines), usually leading to several different statuses.

8.5.3 Execution Steps and Time

The system described in a statechart changes state and executes actions based on execution steps. These execution steps determine a sequence of dynamic changes. They are logical intervals, and are not associated with a particular time. In other words, steps are a series of snapshots of the system's situation, where these snapshots represent the status at the given point of time; changes caused by evaluating a snapshot will be sensed at the next snapshot.

Since statecharts express a reactive system, steps are event-driven, or to be more precise, driven by triggers: events, conditions or a combination thereof. Asynchronous external triggers can cause some reactions in the system (for example, state transitions), which in turn may cause an ordered chain of internal reactions, such as generating events, actions and state transitions as if they were a series of synchronous steps.

At any given step **n**, the system evaluates the events that were generated and identifies the values of data items and conditions that are present at the occurrence of step **n**. Actions are carried out simultaneously after an enabling trigger is detected at step **n**. Action-based calculations are based on status and values at the beginning of a step. The enabling trigger and associated actions that lead into a compound transition are taken and completed at step **n**. All the changes caused by the execution of a transition (step **n**) are sensed in the following step. Analogously, static reactions sensed within a state will be carried out at the next step.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For example, an event that was generated by an action at step **n** will be sensed only in the following step **n**+1. Note that internally generated events have a lifetime of one step only; they are not remembered beyond step **n**+1. Similarly, if an action is defined based on entering (or exiting) a state, or if an activity is to be started within a state, only the step following the transition into the given state detects these. Note that this does not apply to the notion of performing an activity **throughout** a state. In this case, the activity will be started at the transition step entering the state and respectively deactivated at the exiting step.

The fact that the results of an action are only sensed at the following step implies that a state cannot be entered and exited at the same step. However, for a self-looping transition, a state can be exited and reentered in one step.

Note: An action that generates the triggering event for an activity may also modify certain conditions and data items within the same execution step. The results of the modification are visible to the invoked activity.

8.5.4 Synchronisation and Race Conditions

Since a transition can perform a set of actions simultaneously in an arbitrary order, and multiple transitions can be enabled at a single step, due to orthogonal states, race conditions and synchronisation must be considered.

Race conditions arise when the value of an element (condition or data item) is modified more than once, or is both modified and used, in a single step.

If an element is both modified and used in the same step, the value that the element had at the beginning of the step is used to evaluate all expressions that depend on it. A modified result is never used to evaluate expressions in the same step as the modification. Thus, the race condition is resolved.

If an element is to be modified multiple times during a single step (for example, by different assignments to the same data items in concurrent transitions in orthogonal states), the system cannot resolve this nondeterminism and is in an illegal state. The design of the system must prevent these cases.

As defined in Subclause 8.5.3 on page 295, the evaluation of the internal chain of reactions follows the same rules as for detecting and reacting to external events. Actions may generate internal events, which in turn trigger a next step and cause some actions to be performed, and so forth. This provides for ordered behaviour, and is used for synchronisation of concurrent state transitions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.6 Summary of Language Elements

The following subclauses summarise the language elements used in state charts.

8.6.1 Event Expressions

Table 8-1 shows events that are related to other elements.

Table 8-1 Events Related to Other Elements

Event	Occurs when:	Notes
en(S)	State **S** is entered.	—
ex(S)	State **S** is exited.	—
entering	State is entered.	This applies only to the static reactions field in the state definition.
exiting	State is exited.	This applies only to the static reactions field in the state definition.
st(A)	Activity **A** is started.	—
sp(A)	Activity **A** is stopped.	—
ch(V)	The value of data item expression is changed.	**V** is a string or numeric expression.
tr(C)	The value of condition **C** is set to TRUE (from FALSE).	—
fs(C)	The value of condition **C** is set to FALSE(from TRUE).	—
rd(V)	Data item **V** is read.	—
wr(V)	Data item **V** is written.	**V** is a primitive data item.
tm(E,N)	**N** clock units passed from last time event E occurred.	**N** is a numeric expression. Unless noted otherwise, the real timeout value is implementation or policy-dependent.

Table 8-2 on page 298 shows compound events.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 8-2 Compound Events

Event	Occurs when:
E[C]	**E** has occurred and condition **C** is TRUE.
not E	**E** did not occur.
E1 and E2	**E1** and **E2** occurred simultaneously.
E1 or E2	**E1** or **E2**, or both occurred.

The operations are presented in descending priority order. Parentheses are used to alter the evaluation order.

8.6.2 Condition Expressions

Table 8-3 shows conditions related to other elements.

Table 8-3 Conditions Related to Other Elements

Condition	TRUE when:	Notes
in(S)	System is in state **S**.	—
ac(A)	Activity **A** is active.	—
hg(A)	Activity **A** is suspended.	—
EXP1 R EXP2	The value of the expression **EXP1** and **EXP2** satisfy the relation **R**.	When expressions are numeric, **R** may be: =, /=, > or <. When they are strings, R may be: = or /=.

Table 8-4 shows compound conditions.

Table 8-4 Compound Conditions

Condition	TRUE when:
not C	**C** is not TRUE.
C1 and C2	Both **C1** and **C2** are TRUE.
C1 or C2	**C1** or **C2**, or both are TRUE.

The operations are presented in descending priority order. Parentheses are used to alter the evaluation order.

8.6.3 Action Expressions

Table 8-5 on page 299 shows actions related to other elements.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 8-5 Actions Related to Other Elements

Action	Performs:	Notes
E	Generate the event **E**.	**E** is a primitive event.
tr!(C)	Assign TRUE to the condition **C**.	**C** is a primitive.
fs!(C)	Assign FALSE to the condition **C**.	**C** is a primitive.
V:=EXP	Assign the value of **EXP** to the data item **V**.	**V** is a primitive (numeric or string) data item.
st!(A)	Activate the activity **A**.	—
sp!(A)	Terminate the activity **A**.	—
sd!(A)	Suspend the activity **A**.	—
rs!(A)	Resume the activity **A**.	—
rd!(V)	Read the value of data item **V**.	**V** is a primitive data item.
wr!(V)	Write the value of data item **V**.	**V** is a primitive data item.

Table 8-6 on page 300 shows compound actions. The compound actions shown can be nested and combined.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 8-6 Compound Actions

Action	Performs:	Notes
A1; A2	Simultaneously perform the actions **A1** and **A2**.	—
If C then A1 else A2 end if	If condition **C** is TRUE, perform action **A1**, otherwise perform **A2**.	The **else** part is optional.
When E then A1 else A2 end when	If event **E** occurred when the action is issued, perform action **A1**, otherwise perform **A2**.	The **else** part is optional.

8.6.4 Data Item Expressions

8.6.4.1 Atomic Numeric Expressions

Atomic numeric expressions may be one of the following:

- a named numeric (integer or real) data item
- a numeric constant, integer or real.

8.6.4.2 Compound Numeric Expressions

The following compound numeric expression may appear in state charts:

- **EXP1 ** EXP2** (exponentiation)
- **EXP1 * EXP2** (multiplication)
- **EXP1 / EXP2** (division)
- **EXP1 + EXP2** (addition)
- **EXP1 – EXP2** (subtraction)
- **+EXP** (positive)
- **–EXP** (negative).

The operations are presented in descending priority order. Parentheses may be used to alter the evaluation order.

8.6.4.3 String Expressions

String expressions may be one of the following:

- a named string data item.
- a string constant: a sequence of characters between apostrophes (for example, 'ABC'). No operations on strings exist.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Section 9. RPC Protocol Definitions

OSI RPC supports the run-time API and application stubs by executing its protocol in response to the events issued by service primitives (see Section 7) and events generated by the underlying network and received RPC PDUs. This results in actions that may generate corresponding service primitives and the invocation of network services.

The following subclauses specify the RPC protocol via the statechart (see Section 8) descriptive technique. Implementations must conform to the external behaviour exhibited by this model to guarantee both interoperability with other implementations and portability of applications using RPC.

The RPC is designed to operate over a transport layer that offers either a reliable, connection-oriented service (COTS) or a datagram, connectionless service (CLTS), or both types of services. Operation of RPC over other protocols and services is not currently defined by this specification.

The details of the RPC protocol differ depending on the selected transport service. The protocols using COTS and CLTS are described separately in this Section.

9.1 Conformance

Because RPC will be implemented in a variety of hardware, software and user environments, the protocol permits latitude in many areas of behaviour.

The protocol machines described in the following subclauses define this space of allowed behaviours. Implementation structure and policy need not follow the protocol machine organisation and defaults. The externally observed behaviour of an implementation, as viewed from the RPC user interface and the transport interface, must be indistinguishable from some subset of the allowed behaviours determined as follows:

- An implementation must completely support at least one of the two protocols described.
- For a given protocol, it is not allowed to omit any part of the behaviour specified unless explicitly noted otherwise.
- The time interval between an input event and the external stimulus that caused the event is arbitrary, but is subject to two constraints:
  - — There is a partial ordering such that all events generated by a flow of execution are delivered in the order of occurrence with respect to other events generated by that flow of execution.
  - — Execution of concurrent state machines must be scheduled fairly so that each machine has equal opportunity to process any pending events and sufficient time to make progress.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2 RPC Stub to Run-time Protocol Machine Interactions

The following subclauses define the interactions between the protocol machines, which are implemented in the RPC run-time system, and the RPC stubs, as applicable to both connectionless (CL) and connection-oriented (CO) protocols.

9.2.1 Client Protocol Machines

The RPC stub generates the event START_CALL to invoke a new call, which is associated with Call_Handle data. The RPC run-time system dispatches the call, via CO_CLIENT_ALLOC machine for connection-oriented protocol, to the appropriate instance of the protocol machine. The run-time system also sets the conditional flags for the requested execution semantics (IDEMPOTENT, BROADCAST and MAYBE) and the authentication flag AUTH, according to the Call_Handle data structure. If the security service rpc_c_authn_dce_secret is requested and the authentication ticket for this call is already available, the conditional flag TICKET also has to be set to TRUE.

Upon initiating a new RPC session (see Subclause 6.1.3 on page 269) an instance of a client call protocol machine is created (CO_CLIENT or CL_CLIENT). If it is a consecutive call within an already opened association (connection-oriented protocol) or activity (connectionless protocol), the call is dispatched to the appropriate idling client call machine.

The RPC stub may queue the marshalled call data either in one segment or in chunks of segments, depending on the call type (for example, whether a pipe data type is opened) and the local memory management policies. The run-time system detects the availability of data and sets the conditional flag TRANSMIT_REQ to TRUE if data for at least one PDU fragment is available. The run-time system resets TRANSMIT_REQ if the queue contains temporarily less than a PDU fragment of data. The sizes of data segments queued by the stub are not necessarily equivalent to the sizes of PDU fragments sent by the run-time system.

If the transmit queue only contains data for the last PDU fragment to be sent, the RPC run-time system sets the conditional flag LAST_IN_FRAG. Note that if the request is to be a single packet PDU, LAST_IN_FRAG must also be set.

Response data (**out** parameters) are processed at the RPC run-time system in PDU fragment granularity. Each inbound data fragment gets buffered and transferred to the stub through the activity HANDLE_OUT_FRAG. RPC stub implementation policy determines whether it processes incomplete response data. When the client run-time system has received and buffered the complete response, it signals the completion and transfers the control to the stub by raising the event RCV_LAST_OUT_FRAG. Note that the stub must assure that the HANDLE_OUT_FRAG activity has been completed before acting on this event.

Local cancels are transferred to the RPC run-time system by raising the event CLIENT_CANCEL. If an issued cancel was detected by the run-time system, it sets the conditional flag RT_PENDING_CANCEL. To detect cancel requests that may have been issued for a call before the run-time system started execution, the stub transfers this status by setting the conditional flag CURRENT_PENDING_CANCEL along with the START_CALL event. The RT_PENDING_CANCEL status is passed back to the stub after call completion.

If the run-time system terminated the call due to a failure (local or remote), it raises an exception by calling the activity EXCEPTION. The data item RT_EXCEPTION_TYPE indicates the type of failure to the stub, using fault and reject status codes. The conditional flag RT_DID_NOT_EXECUTE further details the execution status of the call (connection-oriented protocol only).

If a context handle is activated, the stub generates a CONTEXT_ACTIVE event and identifies the client/server pair for which this context handle is active. A context handle becomes active when a server returns a value that is not NULL for an RPC context handle parameter. For each context handle that becomes active, the client stub must generate this event.

If a context handle becomes inactive, the stub generates a CONTEXT_INACTIVE event and identifies the client/server pair for which this context handle is no longer active. A context handle becomes inactive when a server returns a NULL value for an RPC context handle parameter. For each context handle that becomes inactive, the client stub must generate this event.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.2.2 Server Protocol Machines

The server call protocol machines (CO_SERVER and CL_SERVER) are instantiated at an RPC request for a call in a new session, which is a new association for the connection-oriented protocol or a new activity for the connectionless protocol. If a session has already been established, the server call machines are idling while waiting to accept new call requests unless a context rundown was issued.

Request data (**in** parameters) are processed at the RPC run-time system in PDU fragment granularity. Each inbound data fragment gets buffered and transferred to the stub through the activity HANDLE_IN_FRAG. RPC stub implementation policy determines whether it processes incomplete request data. When the client run-time system has received and buffered the complete request, it signals the completion and transfers the control to the stub by raising the event RCV_LAST_IN_FRAG. Note that the stub must assure that the HANDLE_IN_FRAG activity has been completed before acting on this event.

When the server application procedure is ready to respond to the RPC request with **out** parameter data, the stub signals this to the run-time system by raising the event PROC_RESPONSE. The called application procedure may not have completed at the time of this event, depending on the call type.

The RPC stub may queue the marshalled call data for the response either in one segment or in chunks of segments, depending on the call type (for example, whether a pipe data type is opened) and the local memory management policies. The run-time system detects the availability of data and sets the conditional flag TRANSMIT_RESP to TRUE if data for at least one PDU fragment is available. The run-time system resets TRANSMIT_RESP if the queue contains temporarily less than a PDU fragment of data. The sizes of data segments queued by the stub are not necessarily equivalent to the sizes of PDU fragments sent by the run-time system.

If the transmit queue only contains data for the last PDU fragment to be sent, the RPC run-time system sets the conditional flag LAST_OUT_FRAG. Note that if the request is to be a single packet PDU, LAST_OUT_FRAG must also be set.

Upon detecting a cancel request issued by the client, the server run-time system starts the activity CANCEL_NOTIFY_APP to notify the stub that a cancel was issued. The stub returns the status RETURN_PENDING_CANCEL to the run-time system after processing the cancel request and terminating the activity CANCEL_NOTIFY_APP.

If the server manager routine rejects the call before execution, the RPC stub signals the run-time system by raising the event PROCESSING_FDNE. If the stub detected a processing failure during execution of the request, it signals the run-time system by raising the event PROCESSING_FAULT.

If a context handle is activated, the stub generates a CONTEXT_ACTIVE event and identifies the client/server pair for which this context handle is active. A context handle becomes active when a server returns a value which is not NULL for an RPC context handle parameter. For each context handle that becomes active, the server stub must generate this event.

If a context handle becomes inactive, the stub generates a CONTEXT_INACTIVE event and identifies the client/server pair for which this context handle is no longer active. A context handle becomes inactive when a server returns a NULL value for an RPC context handle parameter. For each context handle which becomes inactive, the server stub must generate this event.

If communications between a client/server pair are lost and context handles were active, the server protocol machine generates a RUNDOWN_CONTEXT_HANDLES event. For each active context handle associated with that particular client/server pair, the stub calls the corresponding **<type_id>_rundown** routine.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.3 Connection-oriented Protocol

The connection-oriented protocol behaviour is characterised by concurrent protocol machines of the types specified in Section 11. The number of instances of each of these types and the relationships among these instances are dynamic. These relationships and the information exchanged between these entities can be conceptually decomposed into a four-level hierarchy: client/server, association group, association and call.

9.3.1 Client/Server

An RPC implementation may function as both a client and a server concurrently. For modelling purposes, it may be viewed as containing independent client and server state machines. This corresponds to the client/server model described in Section 6.

The client protocol machines support the client interfaces while the server protocol machines support the server interfaces. Invocation of an RPC may establish relationships between instances of the client and server protocol machines at each of the lower levels in the hierarchy.

The protocol and service for each RPC is handled by a corresponding pair of client CALL machine instance and server CALL machine instance. These instances require a communications channel for exchanging PDUs. This communications channel, shared by a client and server, is known as an **association** and is maintained by a corresponding pair of client and server ASSOCIATION machine instances. A series of RPC calls made from client applications to a specific server may utilise the same association. Concurrent RPCs from a client to the same server may take place over different associations. The set of associations between a client and a server is represented by an association group. Association groups are managed by client and server association group machines (CO_CLIENT_GROUP and CO_SERVER_GROUP). The creation and lifetime of these various protocol machines is a function of resource availability, the relationships described in this subclause, external events and local system policy.

Each client may have multiple simultaneous relationships of the form described in this subclause with multiple servers. Similarly, each server may have multiple simultaneous relationships with multiple clients. Precise details of these relationships are specified in the following subclauses.

9.3.2 Association Group

An association group comprises a set of one or more associations (see Subclause 9.4.3 on page 306 for the definition of association) between a client instance and a server. Each client and server pair may share multiple association groups, although this only occurs if there are multiple, distinguishable protocol towers in use concurrently. A server distinguishes among association groups by using a group identifier, which is unique among the active groups within that server instance. A client distinguishes among association groups based on the server primary address (see Subclause 9.4.3 on page 306) and the group identifier chosen by the server.

Association groups support context handle management and facilitate efficient resource management.

9.3.3 Association

An association represents a communications channel that is shared between a client endpoint and a server endpoint. Each association is layered on top of a single transport connection such that associations and transport connections have a one-to-one correspondence. An association adds a security and presentation context negotiation and some other RPC-specific exchanges to the underlying connection. Each association is a member of one association group. An association can support no more than one RPC at a time, including its affiliated cancels. An association may be serially reused to call any of the interfaces resident at that server's endpoint. For each RPC, an association is allocated, the RPC is made, and the association is deallocated when the RPC completes. Attempting to allocate an association may cause new associations, transport connections and association groups to be made, if necessary, within local client and server policy constraints. Local policy also governs the number and lifetime of associations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.3.3.1 Association Management Policy

Each implementation may determine its own association management policy for accepting new associations and disconnecting existing associations subject to the following constraints:

- An association must be maintained while any RPCs are outstanding on it.
- At least one association in an association group must be maintained if one or more context handles are active between the client and server.

Unusual events (for example, user and management abort requests) may cause associations to be aborted at any time. However, this is likely to cause a pending RPC to fail.

9.3.3.2 Primary and Secondary Endpoint Addresses

A primary endpoint address may be a well-known endpoint or a dynamic endpoint that is registered with an endpoint mapper. For the first association established within an association group, a client specifies the primary endpoint address to request a transport connection to a server.

If a server supports concurrent RPCs, then the server returns a secondary address to the client. The secondary address may be the same as the primary address. Whether they differ is a local implementation-dependent matter.

A client uses this secondary address for subsequent transport connection requests to establish additional concurrent associations to the same server. Each subsequent association established using both the secondary address and group identifier of an association group will be directed to the same server. RPCs on any of the associations within an association group are processed by the same server.

If the server does not return a secondary address, the client will permit only a single association for the corresponding association group. The **rpc_server_listen**() call informs the server RPC run-time system whether to allow concurrent RPCs to the same server.

The absence of a secondary address is modelled as a null value in this specification.

9.3.4 Call

After an association has been allocated for an RPC, the CALL protocol machines (see Subclause 9.4 on page 306) manage the exchange of call data between the client and server. These call machines handle, in an orderly fashion, events that may cause abnormal termination of an RPC. The CALL machines indicate to an RPC client application whether the RPC completed successfully, failed but did not execute, or failed with unknown execution status. Pending cancels are signalled to the client and server applications, and orphaned RPCs are indicated to the server applications. Each RPC is identified by a call identifier that is unique for all currently active RPCs within an association group.

9.3.5 Transport Service Requirements

The RPC CO protocol requires a connection-oriented transport service that guarantees reliable sequential delivery of data. This means the transport guarantees that when it delivers data to a transport user, all data previously sent by the remote transport user on that transport connection has been delivered exactly once, unmodified, and in the order it was presented to the transport by the remote sender.

The COTS must provide connection establishment and release, full-duplex data transfer, segmentation and reassembly, flow control and liveness indication.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.4 Connection-oriented Protocol Machines

The moment in time at which each instance of the protocol machines is created depends on the events that trigger the initial transition into a statechart. Similarly, the lifetime of a protocol machine instance is determined by events that cause transition to the terminal state. All machines may be affected by external events. The relationships among instances of these machines are described in the following subclauses.

The client protocol for processing RPCs is described by the CO_CLIENT_ALLOC, CO_CLIENT_GROUP and CO_CLIENT statecharts.

The server protocol for processing RPCs is described by the CO_SERVER_GROUP and CO_SERVER statecharts.

To avoid race conditions among multiple instances of protocol machines attempting to reference the same state variables or issue conflicting events, a synchronisation mechanism is required. The CO_CLIENT_ALLOC protocol machine illustrates how this synchronisation could be implemented via locking. For simplicity, the other protocol machines merely indicate where synchronisation is necessary, but do not explicitly include the locking steps.

9.4.1 CO_CLIENT_ALLOC

An instance of the CO_CLIENT_ALLOC protocol machine is created each time a new RPC is invoked by the **Invoke** service primitive described in Section 7. This attempts either to create a new association or allocate an existing idle association from the association group indicated by the binding for the RPC. The machine terminates when either an association is allocated or the attempt fails.

Behaviour of this machine is affected by the states of, and the events generated by, instances of the ASSOCIATION protocol machine that correspond to associations within the relevant association group.

This machine defines the recommended policy for allocating associations to RPCs. Implementations may choose a different policy for allocating associations and, thus, are not required to conform to this definition. Any algorithm for retrying failed attempts to allocate an association must retry no more frequently than specified here.

This protocol machine generates the following events, which are input events for the related CO_CLIENT machine instance:

- ALLOC_REQ
- CREATE_ASSOCIATION.

9.4.2 CO_CLIENT_GROUP

An instance of the CO_CLIENT_GROUP protocol machine exists for each association group. It is created upon indication that the first association for this group has been established. It terminates when the last association in the group is terminated.

Behaviour of this machine is affected by the states of, and the events generated by, one or more instances of the ASSOCIATION protocol machine that correspond to associations within the relevant association group.

This machine defines the client management of and the protocol for association groups. Implementations are required to conform to the defined behaviour.

9.4.3 CO_CLIENT

The CO_CLIENT statechart defines the protocol machine types for association and call components. An instance of each of the concurrent protocol machines contained in this statechart is created when a client attempts to establish a new association. It terminates when the relevant association is terminated and related termination activities complete. Instances of the concurrent protocol machines within a CO_CLIENT statechart interact via events and state variables. Also, events generated by the relevant instances of CO_CLIENT_GROUP and CO_CLIENT_ALLOC machines affect these protocol machines.

The CO_CLIENT protocol machine generates the following events, which are input events for the related CO_CLIENT_ALLOC machine instance:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- ALLOC_ASSOC_ACK
- ALLOC_ASSOC_NACK
- CREATE_SUCCESS
- CREATE_FAILED.

The CO_CLIENT protocol machine generates the following events, which are input events for the related CO_CLIENT_GROUP machine instance:

- CREATE_GROUP
- ADD_TO_GROUP
- REMOVE_FROM_GROUP.

9.4.3.1 ASSOCIATION

For each association, an instance of the ASSOCIATION protocol machine defines the the client management of, and the protocol for, that association. The contained machine, labeled INIT, manages the initialisation of an association and the corresponding transport connection. Implementations are required to conform to the defined behaviour.

9.4.3.2 CONTROL

An instance of the CONTROL machine manages the reassembly and dispatching of incoming RPC control PDUs for each association. Implementations are required to conform to the described behaviour.

9.4.3.3 CANCEL

An instance of the CANCEL machine manages cancel requests for an RPC. Implementations are required to conform to the described behaviour.

9.4.3.4 CALL

For each RPC, an instance of the CALL protocol machine defines the client service and protocol for that RPC. Implementations are required to conform to the defined behaviour.

The contained machine, labelled DATA, manages the data exchange between the client and server for the RPC. The machine CONFIRMATION handles the response reception.

9.4.4 CO_SERVER_GROUP

An instance of the CO_SERVER_GROUP protocol machine exists for each association group. It is created upon indication that the first association for this group has been established. It terminates when the last association in the group is terminated and any context for remaining context handles can be rundown.

Behaviour of this machine is affected by the states of, and the events generated by, one or more instances of the ASSOCIATION protocol machine that correspond to associations within the relevant association group.

This machine defines the server management of, and the protocol for, association groups. Implementations are required to conform to the defined behaviour.

9.4.5 CO_SERVER

The CO_SERVER statechart defines the protocol machine types for association and call components. An instance of each concurrent protocol machine contained in the CO_SERVER statechart is created upon indication that a new transport connection to the server has been established. It terminates when the relevant association is terminated. Instances of the concurrent protocol machines within a CO_SERVER statechart interact via events and state variables. Also, events generated by the relevant CO_SERVER_GROUP machine instance affect these protocol machines.

The CO_SERVER protocol machine generates the following events, which are input events for the related CO_SERVER_GROUP machine instance:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- ADD_TO_GROUP
- REMOVE_FROM_GROUP.

9.4.5.1 ASSOCIATION

For each association, an instance of the ASSOCIATION protocol machine defines the server management of, and the protocol for, that association. Implementations are required to conform to the defined behaviour.

9.4.5.2 CONTROL

An instance of the CONTROL machine manages the reassembly and dispatching of incoming RPC control PDUs for each association. Implementations are required to conform to the described behaviour.

9.4.5.3 CANCEL

An instance of the CANCEL machine manages the cancel protocol and service for an RPC. Implementations are required to conform to the described behaviour.

9.4.5.4 WORKING

The WORKING machine defines the handling of an RPC by the server, including the orderly clean up of state after an RPC terminates. The WORKING machine contains the CALL machine.

For each RPC, an instance of the CALL protocol machine defines the server management of, and the protocol for, that RPC. Implementations are required to conform to the defined behaviour. The contained machine, labelled DATA, manages the data exchange between the client and server for the RPC. An instance of the CALL protocol machine is created upon receipt of the first fragment of an RPC request.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.5 Connectionless Protocol

The connectionless protocol behaviour is characterised by concurrent protocol machines of the types specified in Section 10. The number of instances of each of these types and the relationships among these instances is dynamic. These relationships and the information exchanged between these entities can be conceptually decomposed into a three-level hierarchy: client/server, Activity and Call.

9.5.1 Client/Server

The most fundamental partitioning of the protocol machines is between the client and server types. This corresponds to the client/server model described in Section 6. The client protocol machines support the client interfaces while the server protocol machines support the server interfaces. Invocation of an RPC may establish relationships between instances of the client and server protocol machines.

Each client may have multiple simultaneous relationships with multiple servers. Similarly, each server may have multiple simultaneous relationships with multiple clients.

9.5.2 Activity

An **activity** corresponds to a client application instance. Multiple activities may exist concurrently for each client. Both the client and server distinguish among activities by a UUID associated with each activity, called the **activity identifier**. At most one RPC may be in progress for an activity. A series of RPCs may occur sequentially for each activity.

9.5.3 Call

The protocol machines for an RPC manage the exchange of call data between the client and server for an activity. These protocol machines handle, in an orderly fashion, events that may cause abnormal termination of an RPC. The call machines indicate to an RPC client application whether the RPC completed successfully, failed but did not execute, or failed with unknown execution status. Pending cancels are signalled to the client and server applications, and orphaned RPCs are indicated to the server applications. Each RPC is identified by an activity identifier and a sequence number. Activity identifiers may not be reused. A sequence number may be reused for a given activity identifier, if the sequence number space is exhausted. If sequence numbers wrap around and are reused, the implementation must assure that these are unambiguous. Less than half the space of sequence numbers may be used for concurrently pending calls.

9.5.4 Maintaining Execution Context and Monitoring Liveness

The execution context of a call is uniquely identified by the client address space identifier (CAS UUID). This UUID identifies a specific client process instance that is maintaining context with servers. Execution context is not directly related to activities. Multiple activities may run within a single execution context. The client and server run-time system implementations maintain a list of active execution contexts (signalled from the stub by the event CONTEXT_ACTIVE or, respectively, by CONTEXT_INACTIVE).

The server stub indicates, via condition flag CONTEXT_REQUEST, whether it needs to know the execution context identifier (RT_CLIENT_EXECUTION_CONTEXT) for the current call.

Run time implementations monitor liveness of maintained execution contexts periodically. The procedure **convc_indy**( ), as specified in Annex P, may be used for liveness monitoring. Compliant implementations must provide the specified conversation manager interface. There are no guarantees about the time periods of liveness indications by clients (the default for invocation of **convc_indy**( ) is 20 seconds), and it is implementation-specific how this operation is used to monitor liveness. The server protocol machine generates the event RUNDOWN_CONTEXT_HANDLES if it determines that it has lost contact with the client (see also Subclause 9.3 on page 304).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.5.5 Serial Numbers

Serial numbers allow data senders to match a **fack** PDU with the **request** or **response** PDU that induced the **fack** PDU to be sent. Serial numbers are used according to the following model. The sender of data maintains a queue of all PDUs that have been sent but not yet acknowledged. The sender also maintains a current serial number, which is initialised to 0 (zero) when a call begins. Each time a data (**request** or **response**) or **ping** PDU is sent or resent from the queue, the current serial number is incremented and inserted into the outgoing data PDU; each PDU in the queue records the serial number used in the most recent transmission of the PDU. When the receiver of a data PDU sends a **fack** PDU in reply, it inserts the serial number of the data PDU into the **fack** body. This is the serial number of the PDU that induced the **fack**.

Upon receiving a **fack** PDU, the data sender must take the following steps:

a) eliminate from its queue all PDUs that are being acknowledged by the **fack** PDU

b) decide which, if any, of the remaining data PDUs should be retransmitted.

In implementing the second step, the following policies are recommended. It is possible that some PDUs that remain in the queue were in transit at the time the **fack** was generated, and thus could not have been acknowledged by the **fack**. It is likely that such PDUs were received after the **fack** was generated, and retransmitting them would waste network bandwidth. The likelihood of such in-transit PDUs increases as network transmission latency increases.

The potentially gratuitous retransmission of data PDUs can be eliminated by considering the serial number in the **fack** and the serial numbers on the data PDUs in the transmit queue. In particular, the data sender should not retransmit any data PDU whose serial number (that is, the serial number used in the most recent transmission of the data PDU) is greater than the serial number in the **fack** PDU.

Note: The recommended policy assumes that the occurrences of spontaneous (network-induced) re-ordering of PDUs is rare.

Because serial numbers allow a transmission and a reply to be matched up, serial numbers can be used in the course of estimating the network round trip time (RTT) between sends and receives. Such an estimate of RTT can be used to control retransmission policy.

9.5.6 Transport Service Requirements

The connectionless protocol requires a connectionless, datagram transport (CLTS). The CLTS must provide a full-duplex datagram service that delivers transport user data on a best effort basis. The CLTS may lose, delay, reorder and duplicate transport service data units. Transport must not misdeliver or modify user data. The CLTS must guarantee that the maximum lifetime of each transport service data unit is bounded.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.6 Connectionless Protocol Machines

The moment in time at which each instance of the protocol machines is created depends upon events that trigger transitions from the initial state. The lifetime of a protocol machine instance is determined by the lifetime of the corresponding activity. All machines may be affected by external events. The relationships among instances of these machines are described in the following subclauses.

The client protocol for processing RPCs is described by the CL_CLIENT statechart.

The server protocol for processing RPCs is described by the CL_SERVER statechart.

9.6.1 RPC Stub to Run Time Protocol Machine Interactions

Since the connectionless RPC protocol machines have to take into account the unreliable nature of the underlying datagram transport, the RPC run-time system has to handle fragmentation, the possible delivery of packets out of order, and the reassembly of the entire request or response data.

In accordance to the semantics of the HANDLE_IN_FRAG and HANDLE_OUT_FRAG activities, the run-time system buffers out-of-order fragments temporarily and makes received fragments available to the stub only if they are consecutive (see Subclause 10.1.1 on page 316 and Subclause 10.2.1 on page 344). The RECEIVED_LAST_IN_FRAG and RECEIVED_LAST_OUT_FRAG events are only generated if the received data is complete; that is, there are no outstanding fragments.

9.6.2 CL_CLIENT

The CL_CLIENT statechart defines the client protocol machine types for an RPC. An instance of each of the protocol machines is created when an **Invoke** service primitive, as defined in Section 7, is first generated for an activity. Subsequent **Invoke** primitives for the same activity are handled by the same instance of the CL_CLIENT statechart. The lifetime of the protocol machines corresponds to that of the associated activity. The concurrent protocol machines for an instance of a CL_CLIENT statechart interact via events and state variables.

9.6.2.1 CONTROL

An instance of the CONTROL machine defines the protocol used to manage the reassembly and dispatching of received control PDUs for each RPC. Implementations must conform to the described behaviour.

9.6.2.2 AUTHENTICATION

An instance of the AUTHENTICATION machine manages the authentication service for each activity. It handles and verifies mutual authentication if a security service is requested for the associated RPC. It is independent of the underlying authentication protocol and the specific protection services that are in use. Implementations are required to conform to the described behaviour.

9.6.2.3 CALLBACK

An instance of the CALLBACK machine defines the protocol used to manage callbacks to the client for an RPC. Implementations must conform to the described behaviour.

9.6.2.4 PING

An instance of the PING machine defines the protocol used to ascertain liveness of the server for each RPC. Implementations must conform to the described behaviour.

9.6.2.5 CANCEL

An instance of the CANCEL machine defines the protocol used to manage cancel requests for each RPC. Implementations must conform to the described behaviour.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.6.2.6 DATA

An instance of the DATA machine defines the client side of the protocol used to manage the data exchange between the client and server for each RPC. The contained machines labelled REQUEST and CONFIRMATION handle the request transmission and response receipt, respectively. Implementations must conform to the described behaviour.

9.6.3 CL_SERVER

The CL_SERVER statechart defines the server protocol machine types for an RPC. An instance of each of the protocol machines is created upon indication that an RPC request PDU for a new activity has been received. Subsequent RPC request PDUs for the same activity are handled by the same instance of the CL_SERVER statechart. Thus, the lifetime of the protocol machines corresponds to that of the associated activity. The concurrent protocol machines for an instance of a CL_SERVER statechart interact via events and state variables.

9.6.3.1 CONTROL

An instance of the CONTROL machine defines the protocol used to manage the reassembly and dispatching of received control PDUs for each RPC. Implementations must conform to the described behaviour.

9.6.3.2 AUTHENTICATION

An instance of the AUTHENTICATION machine manages the authentication service for each activity. It handles and verifies mutual authentication if a security service is requested for the associated RPC. It is independent of the underlying authentication protocol and the specific protection services that are in use. Implementations are required to conform to the described behaviour.

9.6.3.3 CANCEL

An instance of the CANCEL machine defines the protocol used to manage cancels received for each RPC. Implementations must conform to the described behaviour.

9.6.3.4 WORKING

The WORKING machine defines the handling of an RPC by the server, including the orderly clean up of state after an RPC terminates. The WORKING machine contains the CALL machine.

For each RPC, an instance of the CALL protocol machine defines the server management of, and the protocol for, that RPC. The CALL machine is composed of two subordinate machines, DATA and CALLBACK. An instance of the DATA machine defines the server side of the protocol that is used to manage the data exchange between the client and server for each RPC. An instance of the CALLBACK machine defines the protocol used to manage conversation manager callbacks to the client, enabling servers to enforce at-most-once execution semantics.

Implementations are required to conform to the defined behaviour for the WORKING protocol machine and the protocol machines contained within WORKING.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

9.7 Naming Conventions

To provide better readability, many description elements in the protocol machines are named according to the naming conventions described in the following list. Elements are categorised into groups by using common prefixes to their names:

CONST_ An implementation-specific or architected constant value, declared as a data item.

DO_ An action that processes and evaluates PDU contents.

MAX_ A constant that defines a maximum value. A maximum value is either architected, implementation-specific or application-defined.

PDU_ A data item or condition that represents data fields or flags of a currently received PDU.

RCV_ A compound event that indicates the receipt of a particular PDU.

RT_ A run-time (protocol machine) internal data item or condition. These are usually used to preserve state information.

SND_ A data item or condition that represents the values to be sent with the next PDU (input for activity SEND_PKT).

TIMEOUT_ A constant that defines a timeout value. A timeout value is either architected, implementation-specific or application-defined.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 10. Connectionless RPC Protocol Machines

This Section specifies the connectionless RPC protocol as a series of statecharts and accompanying tables of definitions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10.1 CL_CLIENT Machine

Figure 10-1 shows the CL_CLIENT machine statechart.

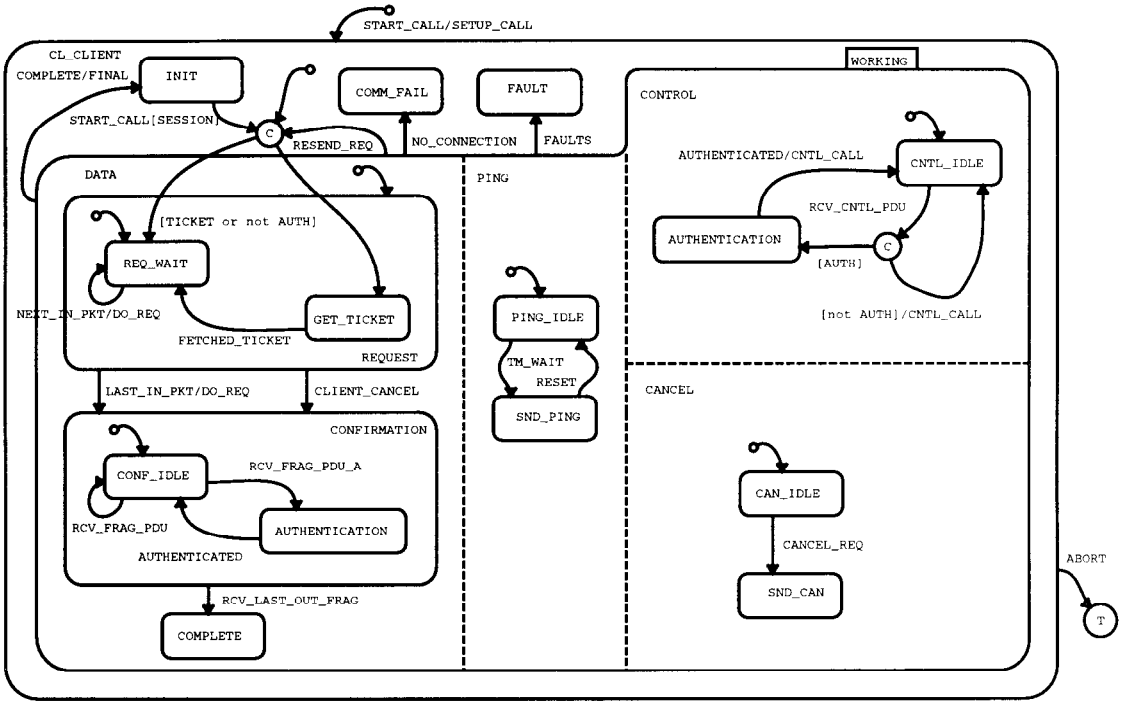


Figure 10-1 CL_CLIENT Statechart

10.1.1 CL_CLIENT Activities

The CL_CLIENT statechart defines the following activities:

Chart: CL_CLIENT

Activity: ABORT_CALL

Description: Flush and discard any further responses for this call. If this activity was caused by a detected failure, there may be numerous additional packets in the pipeline. The flush may be lazy, upon subsequent receive processing. Also, notify the run-time system and stub to reclaim any resources for this call.

Chart: CL_CLIENT

Activity: EXCEPTION

Description: Raise a fault and return to calling routine.

Chart: CL_CLIENT

Activity: FETCH_TICKET

Description: Obtains the security context for the RPC session from the security service (that is, kerberos ticket, if authentication service is rpc_c_authn_dce_secret).

The activity resets the conditional flag TICKET to false at the beginning, and sets TICKET to true before termination only if the fetch operation succeeded. FETCH_TICKET is a self-terminating activity.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT

Activity: HANDLE_OUT_FRAG

Description: This activity is invoked at each received fragment evaluation of **out** parameters for multi-fragmented RPC responses. The HANDLE_OUT_FRAG activity makes received data of continuous fragments available to the stub for unmarshalling and the object UUID (RT_OBJ_ID) available to the manager routine. This does not require a transfer of control from the run-time system to the stub for each fragment; implementation policy determines when control is transferred.

In addition, if the client receives a fragment out of order, HANDLE_OUT_FRAG buffers this fragment temporarily until subsequently received fragments allow for a consecutive inclusion of these temporarily buffered fragments. The fragment ordering is determined by the fragment number (RT_OUT_FRAG_NUM). If previously buffered out of order fragments are appended to the continuous receive buffer (RT_OUT_PARAMS), HANDLE_OUT_FRAG must also adjust the state variable RT_CONT_OUT_FRAG_NUM. HANDLE_OUT_FRAG also maintains the selective acknowledgement bit masks which are used in the **fack** PDU.

Modifications of RT_CONT_OUT_FRAG_NUM must be synchronised with other actions (DO_OUT_PKT) performed by the protocol machine.

Chart: CL_CLIENT

Activity: RESET_IN_FRAG

Description: This activity is invoked if the protocol machine determines that a set of fragments needs to be retransmitted. RESET_IN_FRAG resets the values of SND_FRAG_NUM and RT_IN_FRAG to the beginning of the transmission queue. Note that the condition LAST_IN_FRAG must also be set appropriately.

The first fragment in this queue may not have fragment number 0, since other fragment acknowledgements may have been received, allowing the client to free previously sent data packets. Run time implementations must keep state about the acknowledgement of sent fragmented request PDUs.

This activity generates the event RESEND and marks TRANSMIT_REQ as true to trigger the transitions that actually invoke the appropriate SEND_PKT activities.

Chart: CL_CLIENT

Activity: SEND_PKT

Description: Prepare a PDU to send to the server, adding the appropriate header information as necessary. If security services were requested (conditional flag AUTH is true), apply per-message security services. Send the PDU.

The conditional flags and data items set in the run-time system (with prefix SND_) provide the appropriate input for generating the PDU data. Note that actions within the same execution step that started this activity may have assigned values to the SND_* variables which have to be taken by this instance of the activity.

After sending a request PDU, the RT_IN_FRAG pointer is incremented accordingly, to point to the remaining data in the transmit queue.

Note: The SEND_PKT activity may be invoked simultaneously by several orthogonal states (DATA, CONTROL, CANCEL, and so on). The run-time system must catch these send requests, buffer these and the associated data, and perform the sends in sequential order.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT

Activity: VERIFY_AUTH

Description: Verify the authentication trailer of PDU and decrypt message if necessary.

This activity takes as input values the PDU header field **auth_proto** (authentication protocol specifier: RT_AUTH_SPEC) and the authentication verifier (PDU trailer: RT_AUTH_VERIFIER).

Depending on the result of the verification, the activity VERIFY_AUTH generates either the event AUTHENTICATED (success) or DENIED (authentication failure).

The algorithm applied to this activity is dependent on the security service in use (determined by RT_AUTH_SPEC). The general evaluation steps for authentication service rpc_c_authn_dce_secret are as follows (for more details see Section 13):

- Check the protection level applied to the PDU (parameter in RT_AUTH_VERIFIER) against the protection level for the call (negotiated security context). If matching, proceed with verification, otherwise raise DENIED.
- Decrypt the cyphertext portion of the verifier and verify PDUs integrity. If discrepancies are found, raise DENIED, otherwise raise AUTHENTICATED and proceed (if privacy protected).
- If privacy protection is requested, decrypt PDU body data.

Note: The VERIFY_AUTH activity may be invoked simultaneously by several orthogonal states (DATA, CONTROL and CANCEL). VERIFY_AUTH must not generate the event AUTHENTICATED unless the entire requested authentication processing is completed. If VERIFY_AUTH detects an authentication failure and generates the event DENIED, the protocol machine rejects the RPC call and no further processing is required.

10.1.2 CL_CLIENT States

The CL_CLIENT statechart defines the following states:

Chart: CL_CLIENT

State: AUTHENTICATION

Description: Process authentication verification.

Reactions	
Trigger	**Action**
[RT_SECURITY_CONTEXT]	st!(VERIFY_AUTH)

Chart: CL_CLIENT

State: AUTHENTICATION

Description: Process authentication verification.

Reactions	
Trigger	**Action**
[RT_SECURITY_CONTEXT]	st!(VERIFY_AUTH)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

State: CANCEL

Description: Processing of requests to terminate a call in progress.

Chart: CL_CLIENT

State: CAN_IDLE

Description: Wait for cancel requests.

Reactions	
Trigger	**Action**

en(CAN_IDLE)	IF
	  CURRENT_PENDING_CANCEL
	THEN
	  tr!(RT_PENDING_CANCEL)
	END IF

Chart: CL_CLIENT

State: CL_CLIENT

Description: Main state for statechart CL_CLIENT.

Reactions	
Trigger	**Action**
entering	SND_ACTIVITY_ID:=SESSION_ACTIVITY_ID

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT

State: CNTL_IDLE

Description: Wait for incoming control PDUs.

Reactions	
Trigger	**Action**
en(CNTL_IDLE)	fs!(FAULT_PDU); fs!(WORKING_PDU); fs!(NOCALL_PDU); fs!(REJECT_PDU); fs!(FACK_PDU); fs!(CANCEL_ACK_PDU) T{ RECEIVE_PDU[PDU_TYPE=FACK and VALID_PDU_HEADER]
RECEIVE_PDU[PDU_TYPE=FAULT and VALID_PDU_HEADER]	tr!(FAULT_PDU)
exiting	IF AUTH THEN RT_AUTH_VERIFIER_CNTL:= PDU_AUTH_VERIFIER END IF
RECEIVE_PDU[PDU_TYPE=WORKING and VALID_PDU_HEADER]	tr!(WORKING_PDU)
RECEIVE_PDU[PDU_TYPE=NOCALL and VALID_PDU_HEADER]	tr!(NOCALL_PDU)
RECEIVE_PDU[PDU_TYPE=REJECT and VALID_PDU_HEADER]	tr!(REJECT_PDU)
RECEIVE_PDU[PDU_TYPE=CANCEL_ACK and VALID_PDU_HEADER]	tr!(CANCEL_ACK_PDU); RT_RCV_CANCEL_ID:=PDU_CANCEL_ID
RECEIVE_PDU[CNTL_PDU and VALID_PDU_HEADER]	RCV_CNTL_PDU

Chart: CL_CLIENT

State: COMM_FAIL

Description: Handle communication failures.

Reactions	
Trigger	**Action**
en(COMM_FAIL)	RT_EXCEPTION_STATUS:=CONST_COMM_FAILURE; st!(EXCEPTION); st!(ABORT_CALL)

Chart: CL_CLIENT

State: COMPLETE

Description: Call completed successfully (If NON_IDEMPOTENT wait for ACK_TIMEOUT).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT

State: CONFIRMATION

Description: Process response data (**out** parameters) for remote procedure call.

Reactions	
Trigger	**Action**
en(CONFIRMATION)	fs!(RESPONSE_ACTIVE); RT_OUT_PARAMS:=NULL; RT_OUT_SERIAL_NUM:=-1; RT_CONT_OUT_FRAG_NUM:=-1; RT_LAST_OUT_FRAG_NUM:=-1; fs!(LAST_OUT_FRAG)

Chart: CL_CLIENT

State: CONF_IDLE

Description: Receive response data from server (possibly fragmented).

Reactions	
Trigger	**Action**
en(CONF_IDLE)[RESPONSE_ACTIVE and RT_OUT_FRAG_NUM=RT_CONT_OUT_FRAG_NUM+1]	DO_OUT_PKT; st!(HANDLE_OUT_FRAG)
en(CONF_IDLE)[RESPONSE_ACTIVE and (not NO_FACK or NO_FACK and RT_BUF_LIMIT)]	FACK_CALL
en(CONF_IDLE)[RESPONSE_ACTIVE and RT_OUT_FRAG_NUM/=RT_CONT_OUT_FRAG_NUM+1]	st!(HANDLE_OUT_FRAG)
RECEIVE_PDU[PDU_TYPE=RESPONSE and VALID_PDU_HEADER and not AUTH]	tr!(RESPONSE_ACTIVE); DO_RESP; RCV_FRAG_PDU
RECEIVE_PDU[PDU_TYPE=RESPONSE and VALID_PDU_HEADER and AUTH]	tr!(RESPONSE_ACTIVE); DO_RESP; RCV_FRAG_PDU_A
en(CONF_IDLE)[RESPONSE_ACTIVE and LAST_OUT_FRAG and RT_LAST_OUT_FRAG_NUM= RT_CONT_OUT_FRAG_NUM+1]	RCV_LAST_OUT_FRAG

Chart: CL_CLIENT

State: CONTROL

Description: Process received control PDUs.

Chart: CL_CLIENT

State: DATA

Description: Process RPC call data.

Chart: CL_CLIENT

State: FAULT

Description: Handle faults in processing call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Reactions	
Trigger	**Action**
en(FAULT)[FAULT_PDU]	RT_EXCEPTION_STATUS:=PDU_FAULT_STATUS; st!(EXCEPTION); st!(ABORT_CALL)
en(FAULT)[REJECT_PDU]	RT_EXCEPTION_STATUS:=PDU_REJECT_STATUS; st!(EXCEPTION); st!(ABORT_CALL)
en(FAULT)[not FAULT_PDU and not REJECT_PDU]	RT_EXCEPTION_STATUS:=PDU_REJECT_STATUS; st!(EXCEPTION); st!(ABORT_CALL)

Chart: CL_CLIENT

State: GET_TICKET

Description: Get authentication ticket from security server (security service-specific).

Activities Throughout:
FETCH_TICKET

Chart: CL_CLIENT

State: INIT

Description: Initial remote procedure call state.

Reactions	
Trigger	**Action**
exiting	SETUP_CALL

Chart: CL_CLIENT

State: PING

Description: Main state to handle asynchronous ping requests.

Chart: CL_CLIENT

State: PING_IDLE

Description: Wait for expiration of WAIT_TIMEOUT.

Reactions	
Trigger	**Action**
exiting	RT_PING_COUNT:=0

Chart: CL_CLIENT

State: REQUEST

Description: Process request data (**in** parameters) for remote procedure call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT

State: REQ_WAIT

Description: Handle fragmented requests to server.

Reactions	
Trigger	**Action**
tm(en(REQ_WAIT), TIMEOUT_FRAG)	RESEND_IN_FRAGS
en(REQ_WAIT)[not REQUEST_ACTIVE]	FIRST_REQ

Chart: CL_CLIENT

State: SND_CAN

Description: Process cancel requests. Sends cancel PDU.

Reactions	
Trigger	**Action**
en(SND_CAN) or CLIENT_CANCEL or tm(CLIENT_CANCEL, TIMEOUT_CANCEL)	CAN_CALL

Chart: CL_CLIENT

State: SND_PING

Description: Send **ping** PDU.

Reactions	
Trigger	**Action**
en(SND_PING) or tm(en(SND_PING), TIMEOUT_PING)	RT_PING_COUNT:=RT_PING_COUNT+1; SND_SERIAL_NUM:=SND_SERIAL_NUM+1; SND_REQUEST_TYPE:=PING; st!(SEND_PKT)

Chart: CL_CLIENT

State: WORKING

Description: Main working state for call instance.

10.1.3 CL_CLIENT Events

The CL_CLIENT statechart defines the following events:

Chart: CL_CLIENT

Event: ABORT

Description: RPC session (same activity UUID) has terminated.

Definition: `st(ABORT_CALL) or sp(FETCH_TICKET)[not TICKET]`

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Event: AUTHENTICATED

Description: Authentication processing completed successfully.

Chart: CL_CLIENT

Event: CANCEL_REQ

Description: Events which cause a transition into an active cancel state.

Definition:
```
CLIENT_CANCEL or [RT_PENDING_CANCEL] or
 tm(CLIENT_CANCEL, TIMEOUT_CANCEL)
```

Chart: CL_CLIENT

Event: CLIENT_CANCEL

Description: The client has issued a request to terminate a call.

Chart: CL_CLIENT

Event: COMPLETE

Description: RPC completed (with success or fault).

Definition:
```
(en(COMPLETE)[not NON_IDEMPOTENT] or
 en(CONFIRMATION)[MAYBE] or TM_ACK) and not
 NO_CONNECTION and not FAULTS
```

Chart: CL_CLIENT

Event: DENIED

Description: Authentication failure detected.

The VERIFY_AUTH activity generates this event if either the integrity check failed or the requested protection level for authentication services does not match.

Chart: CL_CLIENT

Event: FAULTS

Description: Received a **fault** or **reject** PDU or a PDU with wrong authentication verifier.

Definition:
```
(RCV_FAULT or DENIED) and not NO_CONNECTION
```

Chart: CL_CLIENT

Event: FETCHED_TICKET

Description: Client fetched a valid Ticket Granting Ticket.

Definition:
```
sp(FETCH_TICKET)[TICKET]
```

Chart: CL_CLIENT

Event: LAST_IN_PKT

Description: Statechart internal event: last packet of fragmented request.

Definition:
```
[TRANSMIT_REQ and LAST_IN_FRAG and BURST and
 REQUEST_ACTIVE] or
 en(CNTL_IDLE)[TRANSMIT_REQ and FACK_PDU
 and LAST_IN_FRAG and IN_FRAG_NUM_EQ and
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
REQUEST_ACTIVE]
```

Chart: CL_CLIENT

Event: NEXT_IN_PKT

Description: Statechart internal event: intermediate packet of fragmented request.

Definition:
```
[TRANSMIT_REQ and not LAST_IN_FRAG and BURST
 and REQUEST_ACTIVE] or
 en(CNTL_IDLE)[TRANSMIT_REQ and FACK_PDU
 and not LAST_IN_FRAG and IN_FRAG_NUM_EQ and
 REQUEST_ACTIVE]
```

Chart: CL_CLIENT

Event: NO_CONNECTION

Description: Detected communications failure due to timeout events or excessive retries.

Definition:
```
tm(en(CONFIRMATION),
 TIMEOUT_BROADCAST)[BROADCAST] or
 [in(SND_PING) and
 RT_PING_COUNT>MAX_PINGS] or
 [RT_REQUEST_COUNT>MAX_REQUESTS]
```

Chart: CL_CLIENT

Event: RCV_CNTL_PDU

Description: Received one of the control PDUs with valid header.

Chart: CL_CLIENT

Event: RCV_FAULT

Description: Received a **fault** or **reject** PDU. Generated in CNTL_CALL action.

Chart: CL_CLIENT

Event: RCV_FRAG_PDU

Description: Received a **response** PDU for a non-authenticated call.

Chart: CL_CLIENT

Event: RCV_FRAG_PDU_A

Description: Received a **response** PDU for an authenticated call.

Chart: CL_CLIENT

Event: RCV_LAST_OUT_FRAG

Description: Received last fragment of **response** PDU and signalled completion to stub.

The last fragment of a multi-fragmented response or a single packet response was received. RCV_LAST_OUT_FRAG signals that the complete response data is available to the stub for unmarshalling.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Event: RECEIVE_PDU
Description: Received a PDU from server.

Chart: CL_CLIENT
Event: RESEND
Description: Statechart internal event that triggers a resend of previously sent request PDUs.

Chart: CL_CLIENT
Event: RESEND_REQ
Description: Resend the request if no fault was detected.
Definition:
```
RESEND and not NO_CONNECTION and not FAULTS
```

Chart: CL_CLIENT
Event: RESET
Description: Reset ping processing after receiving an acknowledge from server.
Definition:
```
en(CNTL_IDLE)[WORKING_PDU] or
 en(CONF_IDLE) or ex(CONFIRMATION)
```

Chart: CL_CLIENT
Event: START_CALL
Description: Client has initiated an RPC and allocated the data (INVOKE service primitive).

Chart: CL_CLIENT
Event: TM_ACK
Description: Timeout for sending an acknowledge PDU for non-idempotent calls.
Definition:
```
tm(ex(CONFIRMATION),
 TIMEOUT_ACK)[NON_IDEMPOTENT]
```

Chart: CL_CLIENT
Event: TM_WAIT
Description: Timeout for receiving a **response** PDU.
Definition:
```
tm(en(CONF_IDLE) or
 en(PING_IDLE)[in(CONF_IDLE)],
 TIMEOUT_WAIT)
```

10.1.4 CL_CLIENT Conditions

The CL_CLIENT statechart defines the following conditions:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT

Condition: AUTH

Description: Statechart internal flag: indicates that call is authenticated.

Chart: CL_CLIENT

Condition: BOOT_TIME_EQ

Description: Statechart internal flag.

Definition:
```
SND_BOOT_TIME=PDU_BOOT_TIME or
 SND_BOOT_TIME=0
```

Chart: CL_CLIENT

Condition: BROADCAST

Description: Statechart internal flag: broadcast call semantic.

Chart: CL_CLIENT

Condition: BURST

Description: Run time internal flag set if no **fack** is expected before sending next fragment. This flag is used by RPC run-time implementations to optimise the frequency of fragmented outbound packets.

The algorithms used to optimise traffic and avoid congestion are implementation-specific. The protocol machine waits for incoming **fack** PDUs if burst mode is off. The next outbound fragment is triggered by an inbound **fack** PDU.

Run time implementations are responsible for setting the corresponding **nofack** flags in the PDU header.

Chart: CL_CLIENT

Condition: CANCEL_ACK_PDU

Description: Statechart internal flag: received **cancel_ack** PDU.

Chart: **CL_CLIENT**

Condition: CNTL_PDU

Description: Statechart internal flag: control PDUs to be received.

Definition:
```
PDU_TYPE=FAULT or PDU_TYPE=WORKING or
 PDU_TYPE=NOCALL or PDU_TYPE=REJECT or
 PDU_TYPE=FACK
```

Chart: CL_CLIENT

Condition: CURRENT_PENDING_CANCEL

Description: Cancel pending state passed from stub during initialisation of call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Condition: FACK_PDU

Description: Statechart internal flag: received **fack** PDU.

Chart: CL_CLIENT

Condition: FAULT_PDU

Description: Statechart internal flag: received **fault** PDU.

Chart: CL_CLIENT

Condition: IDEMPOTENT

Description: Statechart internal flag: idempotent call.

Chart: CL_CLIENT

Condition: IN_FRAG_NUM_EQ

Description: Statechart internal flag: received frag at server and last sent frag are equal.

This condition verifies the fragment number that was received in a **fack** PDU. (See Section 12 for details.)

Definition: `SND_FRAG_NUM=PDU_FRAG_NUM`

Chart: CL_CLIENT

Condition: IN_FRAG_NUM_NE

Description: Statechart internal flag: received frag at server and last sent frag are not equal.

This condition verifies the fragment number that was received in a **fack** PDU. (See Section 12 for details.)

Definition: `SND_FRAG_NUM/=PDU_FRAG_NUM`

Chart: CL_CLIENT

Condition: LAST_IN_FRAG

Description: Statechart internal flag: last **in** fragment or non-frag **in** packet ready to send.

This flag is set by the run-time system if the transmit queue contains the last fragment (see also Subclause 9.3 on page 304).

Chart: CL_CLIENT

Condition: LAST_OUT_FRAG

Description: Statechart internal flag: last **out** fragment or non-frag **out** packet received.

Chart: CL_CLIENT

Condition: MAYBE

Description: Statechart internal flag: **maybe** call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Condition: NOCALL_PDU
Description: Statechart internal flag: received **nocall** PDU.

Chart: CL_CLIENT
Condition: NON_IDEMPOTENT
Description: Statechart internal flag: non-idempotent (at-most-once) call.
Definition: `not IDEMPOTENT and not BROADCAST and not MAYBE`

Chart: CL_CLIENT
Condition: NO_FACK
Description: Statechart internal flag: received PDU with **nofack** flag true.

Chart: CL_CLIENT
Condition: PDU_FRAG
Description: PDU flag fragment.

Chart: CL_CLIENT
Condition: PDU_LAST_FRAG
Description: PDU flag **lastfrag**.

Chart: CL_CLIENT
Condition: PDU_NO_FACK
Description: PDU flag **nofack**.

Chart: CL_CLIENT
Condition: REJECT_PDU
Description: Statechart internal flag: received **reject** PDU.

Chart: CL_CLIENT
Condition: REQUEST_ACTIVE
Description: Statechart internal flag: send request has started.

Chart: CL_CLIENT
Condition: RESPONSE_ACTIVE
Description: Statechart internal flag: indicates availability of response data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Condition: RT_BUF_LIMIT

Description: Statechart internal flag: buffer limit reached for **out** packets.

The conditional flag RT_BUF_LIMIT triggers the generation of a **fack** PDU which requests the sender of data fragments to readjust the transmission rate.

It is a mechanism to indicate internal buffer limits (overflow) for avoidance of congestions and retransmissions. Since recipients may not evaluate the **fack** body data in a certain way, implementations must not rely on changes in the transmission rate. This indication is an advisory.

Run time implementations are responsible for setting the RT_BUF_LIMIT flag, according to its policies.

Chart: CL_CLIENT

Condition: RT_PENDING_CANCEL

Description: Statechart internal: cancel pending state at server's provider.

Chart: CL_CLIENT

Condition: RT_SECURITY_CONTEXT

Description: Statechart internal flag: set true if security context for call has been established.

Chart: CL_CLIENT

Condition: SEQ_NUM_GTE

Description: Statechart internal flag: received sequence number ≥ initial call sequence number.

Definition: `PDU_SEQ_NUM>=RT_SEQ_NUM`

Chart: CL_CLIENT

Condition: SESSION

Description: Verify that call request is for same session (activity UUID matches).

Definition: `SND_ACTIVITY_ID=SESSION_ACTIVITY_ID`

Chart: CL_CLIENT

Condition: SND_FRAG

Description: Statechart internal flag: header flag **frag** of next fragments to be sent.

Chart: CL_CLIENT

Condition: SND_LAST_FRAG

Description: Statechart internal flag: header flag lastfrag for PDU to be sent.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Condition: TICKET

Description: The authentication ticket is valid (not expired or about to expire).

The authentication ticket from the call's client principal to the server's principal is valid. The particular ticket depends on the client/server pair of principals, and may be different for different RPCs.

Note that implementations may cache unexpired tickets, even across process invocations or system reboots. Therefore, this condition predicate may be maintained external to the RPC run-time system.

Chart: CL_CLIENT

Condition: TRANSMIT_REQ

Description: One or more fragments queued for transmission of request data.

This flag indicates that one or more request fragment(s) are queued in a run-time internal buffer and ready to be transmitted. In conjunction with the BURST flag and possibly expected **fack** PDUs, an event for transmitting the next fragment will be generated.

The run-time system internally sets this flag to true after the stub initially provided data in the transmit queue, sufficient for at least the first PDU fragment to be transmitted. The protocol machine resets this flag if it has detected and taken an event for sending the next fragment in the queue. The run-time system sets this flag again after completion of a SEND_PKT activity if the transmit queue contains enough data for the next PDU fragment to be transmitted.

Chart: CL_CLIENT

Condition: VALID_PDU_HEADER

Description: Pre-evaluation of PDU header (before authentication processing).

Definition:
```
PDU_ACTIVITY_ID=SESSION_ACTIVITY_ID and
 PDU_AUTH_SPEC=RT_AUTH_SPEC and
 SEQ_NUM_GTE and BOOT_TIME_EQ and
 PDU_VERSION_NUM=CL_VERSION_NUM_V20
```

Chart: CL_CLIENT

Condition: WORKING_PDU

Description: Statechart internal flag: received **working** PDU.

10.1.5 CL_CLIENT Actions

The CL_CLIENT statechart defines the following actions:

Chart: CL_CLIENT

Action: CAN_CALL

Description: Set up **cancel** PDU to be sent.

Definition:
```
INCR_CANCEL_ID;
SND_REQUEST_TYPE:=CANCEL;
st!(SEND_PKT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Action: CNTL_CALL

Description: Reactions on received control PDUs.

Definition:

```
IF
  CANCEL_ACK_PDU and
   RT_RCV_CANCEL_ID>RT_CANCEL_ID
THEN
  RT_CANCEL_ID:=RT_RCV_CANCEL_ID
END IF;
IF
  FACK_PDU or NOCALL_PDU and
   PDU_FACK_BODY/=NULL
THEN
  EVAL_FACK_BODY
END IF;
IF
  WORKING_PDU
THEN
  RT_WAIT_COUNT:=RT_WAIT_COUNT+1
END IF;
IF
  NOCALL_PDU or FACK_PDU and
   IN_FRAG_NUM_NE
THEN
  RESEND_IN_FRAGS
END IF;
IF
  FAULT_PDU or REJECT_PDU
THEN
  RCV_FAULT
END IF
```

Chart: CL_CLIENT

Action: DO_OUT_PKT

Description: Append received **response** PDU body data to internal buffer.

Definition:

```
RT_CONT_OUT_FRAG_NUM:=RT_CONT_OUT_FRAG_NUM+1;
RT_OUT_PARAMS:=RT_OUT_PARAMS+RT_BODY
```

Chart: CL_CLIENT

Action: DO_REQ

Description: Send last **in** fragment to server.

Definition:

```
fs!(TRANSMIT_REQ);
IF
  LAST_IN_FRAG
THEN
  tr!(SND_LAST_FRAG)
ELSE
  fs!(SND_LAST_FRAG)
END IF;
SND_FRAG_NUM:=SND_FRAG_NUM+1;
SND_SERIAL_NUM:=SND_SERIAL_NUM+1;
RT_REQUEST_COUNT:=0;
RT_WAIT_COUNT:=0;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
SND_IN_PARAMS:=RT_IN_FRAG;
SND_REQUEST_TYPE:=REQUEST;
st!(SEND_PKT)
```

Chart: CL_CLIENT

Action: DO_RESP

Description: Evaluate **response** PDU header.

Definition:

```
RT_BODY:=PDU_BODY;
RT_OUT_FRAG_NUM:=PDU_FRAG_NUM;
RT_OUT_SERIAL_NUM:=PDU_SERIAL_NUM;
IF
  AUTH
THEN
  RT_AUTH_VERIFIER_CALL:=PDU_AUTH_VERIFIER
END IF;
IF
  PDU_NO_FACK
THEN
  tr!(NO_FACK)
ELSE
  fs!(NO_FACK)
END IF;
IF
  PDU_LAST_FRAG or not PDU_FRAG
THEN
  tr!(LAST_OUT_FRAG);
  RT_LAST_OUT_FRAG_NUM:=PDU_FRAG_NUM
END IF
```

Chart: CL_CLIENT

Action: EVAL_FACK_BODY

Description: Invoke implementation-specific activity to evaluate **fack** PDU body data.

This action reads the **fack** PDU body data according to the PDU specification. It is RPC run-time system implementation-specific how this data will be evaluated and used for subsequent fragmented transmissions. Note that this action also handles **nocall** PDUs that have body data, equivalent to **fack** PDU body data.

Definition:

```
rd!(PDU_FACK_BODY)
```

Chart: CL_CLIENT

Action: FACK_CALL

Description: Send **fack** PDU if **nofack** flag is false or receiver has buffer full condition.

Definition:

```
SND_IN_PARAMS:=RT_FACK_BODY;
SND_REQUEST_TYPE:=FACK;
st!(SEND_PKT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Action: FINAL

Description: Send **ack** PDU to server (for non-idempotent calls only).

Definition:
```
WHEN
  TM_ACK
THEN
  SND_REQUEST_TYPE:=ACK;
  st!(SEND_PKT)
END WHEN
```

Chart: CL_CLIENT

Action: FIRST_REQ

Description: Set up and send first **request** PDU.

If the request is non-fragmented (single PDU), the actual send activity will be performed through the LAST_REQ action.

Definition:
```
tr!(REQUEST_ACTIVE);
SND_FRAG_NUM:=0;
RT_IN_FRAG:=RT_IN_PARAMS;
SND_SEQ_NUM:=RT_SEQ_NUM;
SND_IF_ID:=RT_IF_ID;
SND_IF_VERSION:=RT_IF_VERSION;
SND_OBJ_ID:=RT_OBJ_ID;
SND_OP_NUM:=RT_OP_NUM;
SND_AUTH_SPEC:=RT_AUTH_SPEC;
RT_WAIT_COUNT:=0;
IF
  not LAST_IN_FRAG
THEN
  fs!(TRANSMIT_REQ);
  tr!(SND_FRAG);
  SND_IN_PARAMS:=RT_IN_PARAMS;
  SND_REQUEST_TYPE:=REQUEST;
  st!(SEND_PKT)
ELSE
  fs!(SND_FRAG)
END IF
```

Chart: CL_CLIENT

Action: INCR_CANCEL_ID

Description: Increment **cancel_id** (implementation-specific algorithm).

Definition:
```
RT_CANCEL_ID:=RT_CANCEL_ID+1
```

Chart: CL_CLIENT

Action: INIT_CANCEL_ID

Description: Initialise the cancel_id to be sent in the 1st request (implementation-specific).

Definition:
```
RT_CANCEL_ID:=0;
RT_RCV_CANCEL_ID:=0
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT

Action: RESEND_IN_FRAGS

Description: Perform a resend of previously sent request fragments.

Definition:
```
fs!(TRANSMIT_REQ);
RT_REQUEST_COUNT:=RT_REQUEST_COUNT+1;
st!(RESET_IN_FRAG)
```

Chart: CL_CLIENT

Action: SETUP_CALL

Description: Set up and initialise call data.

Definition:
```
fs!(REQUEST_ACTIVE);
RT_SEQ_NUM:=RT_SEQ_NUM+1;
RT_REQUEST_COUNT:=0;
fs!(SND_LAST_FRAG);
SND_SERIAL_NUM:=0;
INIT_CANCEL_ID;
fs!(RT_PENDING_CANCEL)
```

10.1.6 CL_CLIENT Data-Items

The CL_CLIENT statechart defines the following data items:

Chart: CL_CLIENT

Data Item: ACK

Description: Constant: PDU type **ack**.

Definition: 7

Chart: CL_CLIENT

Data Item: CANCEL

Description: Constant: PDU type **cancel**.

Definition: 8

Chart: CL_CLIENT

Data Item: CANCEL_ACK

Description: Constant: PDU type **cancel_ack**.

Definition: 10

Chart: CL_CLIENT

Data Item: CL_VERSION_NUM_V20

Description: Constant: RPC protocol version 2.0 version number.

Definition: 4

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: CONST_COMM_FAILURE
Description: Reject status code.

Chart: CL_CLIENT
Data Item: FACK
Description: Constant: PDU type **fack**.
Definition: 9

Chart: CL_CLIENT
Data Item: FAULT
Description: Constant: PDU type **fault**.
Definition: 3

Chart: CL_CLIENT
Data Item: MAX_PINGS
Description: Constant for max numbers of unacknowledged pings.

Chart: CL_CLIENT
Data Item: MAX_REQUESTS
Description: Constant for maximum numbers of requests that should be sent per call.

Chart: CL_CLIENT
Data Item: NOCALL
Description: Constant: PDU type **nocall**.
Definition: 5

Chart: CL_CLIENT
Data Item: PDU_ACTIVITY_ID
Description: PDU header field: **act_id**.

Chart: CL_CLIENT
Data Item: PDU_AUTH_SPEC
Description: PDU header field: **auth_proto**.

Chart: CL_CLIENT
Data Item: PDU_AUTH_VERIFIER
Description: PDU trailer: authentication verifier (authentication protocol-specific).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: PDU_BODY
Description: Array of PDU body data.

Chart: CL_CLIENT
Data Item: PDU_BOOT_TIME
Description: PDU header field: **server_boot** (value of zero at first request from client).

Chart: CL_CLIENT
Data Item: PDU_CANCEL_ID
Description: **cancel_id** of received **cancel_ack** PDU body data.

Chart: CL_CLIENT
Data Item: PDU_FACK_BODY
Description: Body information of **fack** PDU (implementation-dependent).

Chart: CL_CLIENT
Data Item: PDU_FAULT_STATUS
Description: Fault status associated with the **fault** PDU body.

Chart: CL_CLIENT
Data Item: PDU_FRAG_NUM
Description: PDU header field: **fragnum**.

Chart: CL_CLIENT
Data Item: PDU_REJECT_STATUS
Description: Reject status code associated with **reject** PDU body.

Chart: CL_CLIENT
Data Item: PDU_SEQ_NUM
Description: PDU header field: **seqnum**.

Chart: CL_CLIENT
Data Item: PDU_SERIAL_NUM
Description: PDU header field: **serial_hi**.

Chart: CL_CLIENT
Data Item: PDU_TYPE
Description: PDU header field: **ptype**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: PDU_VERSION_NUM
Description: PDU header field: **rpc_vers**.

Chart: CL_CLIENT
Data Item: PING
Description: Constant: PDU type **ping**.
Definition: 1

Chart: CL_CLIENT
Data Item: REJECT
Description: Constant: PDU type **reject**.
Definition: 6

Chart: CL_CLIENT
Data Item: REQUEST
Description: Constant: PDU type **request**.
Definition: 0

Chart: CL_CLIENT
Data Item: RESPONSE
Description: Constant: PDU type **response**.
Definition: 2

Chart: CL_CLIENT
Data Item: RT_AUTH_SPEC
Description: Statechart internal: authentication protocol specifier used in current call.

Chart: CL_CLIENT
Data Item: RT_AUTH_VERIFIER_CALL
Description: Statechart internal: received authentication trailer (verifier) for response PDU.

Chart: CL_CLIENT
Data Item: RT_AUTH_VERIFIER_CNTL
Description: Received authentication trailer (verifier) for control PDU.

Chart: CL_CLIENT
Data Item: RT_BODY
Description: Statechart internal: temporarily buffered **response** PDU body data.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_CLIENT
Data Item: RT_CANCEL_ID
Description: Statechart internal: **cancel_id** as received with **cancel_ack** PDU.

Chart: CL_CLIENT
Data Item: RT_CONT_OUT_FRAG_NUM
Description: Statechart internal: last fragment number of continuously buffered **out** block.

Chart: CL_CLIENT
Data Item: RT_EXCEPTION_STATUS
Description: Statechart internal: status value passed to exception handler.

Chart: CL_CLIENT
Data Item: RT_FACK_BODY
Description: Statechart internal: body data for **fack** PDU.

Chart: CL_CLIENT
Data Item: RT_IF_ID
Description: Statechart internal: buffered interface UUID of RPC.

Chart: CL_CLIENT
Data Item: RT_IF_VERSION
Description: Statechart internal: buffered interface version of RPC.

Chart: CL_CLIENT
Data Item: RT_IN_FRAG
Description: Statechart internal pointer to data to be sent in next **request** PDU.

The SEND_PKT activity increments this pointer after a **request** PDU was sent.

Chart: CL_CLIENT
Data Item: RT_IN_PARAMS
Description: Statechart internal: buffered array of reassembled input data.

RT_IN_PARAMS is the queue of transmit data provided by the stub. A possible segmentation of this queue is not equivalent to the sizes of PDU fragments sent by the run-time system (SEND_PKT) activity.

The RT_IN_FRAG variable is a pointer data type that points to the to be transmitted data fragment within this RT_IN_PARAMS queue.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: RT_LAST_OUT_FRAG_NUM
Description: Fragment number of last **out** fragment of remote procedure call.

Chart: CL_CLIENT
Data Item: RT_OBJ_ID
Description: Statechart internal: buffered object UUID of RPC.

Chart: CL_CLIENT
Data Item: RT_OP_NUM
Description: Statechart internal: buffered operation number of RPC.

Chart: CL_CLIENT
Data Item: RT_OUT_FRAG_NUM
Description: Statechart internal: **fragnum** of currently received **response** PDU.

Chart: CL_CLIENT
Data Item: RT_OUT_PARAMS
Description: Buffered array of unfragmented output data.

Chart: CL_CLIENT
Data Item: RT_OUT_SERIAL_NUM
Description: Serial number of sent fragment.

Chart: CL_CLIENT
Data Item: RT_PING_COUNT
Description: Counter for transmitted **ping** PDUs per WAIT cycle.

Chart: CL_CLIENT
Data Item: RT_RCV_CANCEL_ID
Description: Statechart internal: received cancel identifier.

Chart: CL_CLIENT
Data Item: RT_REQUEST_COUNT
Description: The number of times a **request** PDU has been sent for the current fragment.

Chart: CL_CLIENT
Data Item: RT_SEQ_NUM
Description: Sequence number of call: determined by the run-time system (implementation-specific).

The SETUP_CALL action increments this sequence number for every new instance of a call. Implementations may choose a different algorithm, complying to the definition of sequence numbers as specified in Subclause 12.5 on page 468.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: RT_WAIT_COUNT
Description: Statechart internal: counter to determine the length of wait in REQ_WAIT state.

Chart: CL_CLIENT
Data Item: SESSION_ACTIVITY_ID
Description: Statechart internal: activity UUID of current RPC (passed from stub).

Chart: CL_CLIENT
Data Item: SND_ACTIVITY_ID
Description: Activity UUID of current RPC.

Chart: CL_CLIENT
Data Item: SND_AUTH_SPEC
Description: Authentication specifier used for current RPC.

Chart: CL_CLIENT
Data Item: SND_BOOT_TIME
Description: Boot time value promoted to SEND_PKT activity.

Chart: CL_CLIENT
Data Item: SND_FRAG_NUM
Description: Fragment number of PDU to be sent.

Chart: CL_CLIENT
Data Item: SND_IF_ID
Description: Interface UUID of current RPC.

Chart: CL_CLIENT
Data Item: SND_IF_VERSION
Description: Interface version number of current RPC.

Chart: CL_CLIENT
Data Item: SND_IN_PARAMS
Description: PDU body data promoted to SEND_PKT activity.

Chart: CL_CLIENT
Data Item: SND_OBJ_ID
Description: Object UUID of current RPC.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: SND_OP_NUM
Description: Operation number of current RPC.

Chart: CL_CLIENT
Data Item: SND_REQUEST_TYPE
Description: PDU type to be sent.

Chart: CL_CLIENT
Data Item: SND_SEQ_NUM
Description: Sequence number of current RPC.

Chart: CL_CLIENT
Data Item: SND_SERIAL_NUM
Description: Serial number of PDU to be sent.

Chart: CL_CLIENT
Data Item: TIMEOUT_ACK
Description: Timeout value for how long the client will wait before sending an **ack** PDU.

Chart: CL_CLIENT
Data Item: TIMEOUT_BROADCAST
Description: Timeout value for how long the client will wait for response to a broadcast PDU.

Chart: CL_CLIENT
Data Item: TIMEOUT_CANCEL
Description: Timeout value for cancel requests.

Sets the lower bound on the time to wait before timing out after forwarding a **cancel** PDU to the server. The default of this timeout value is set to one second (refer to Annex K.) Applications may set a different value via the **rpc_mgmt_set_cancel_timeout** RPC API.

Chart: CL_CLIENT
Data Item: TIMEOUT_FRAG
Description: Timeout value for wait for a **fack** PDU after a **request** PDU (no **nofack**) sent.

Chart: CL_CLIENT
Data Item: TIMEOUT_PING
Description: Timeout value for how long to wait for response to a **ping** PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_CLIENT
Data Item: TIMEOUT_WAIT
Description: Timeout value for how long the client will wait for a response.

Chart: CL_CLIENT
Data Item: WORKING
Description: Constant: PDU type **working**.
Definition: 4

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

10.2 CL_SERVER Machine

Figure 10-2 shows the CL_SERVER machine statechart.

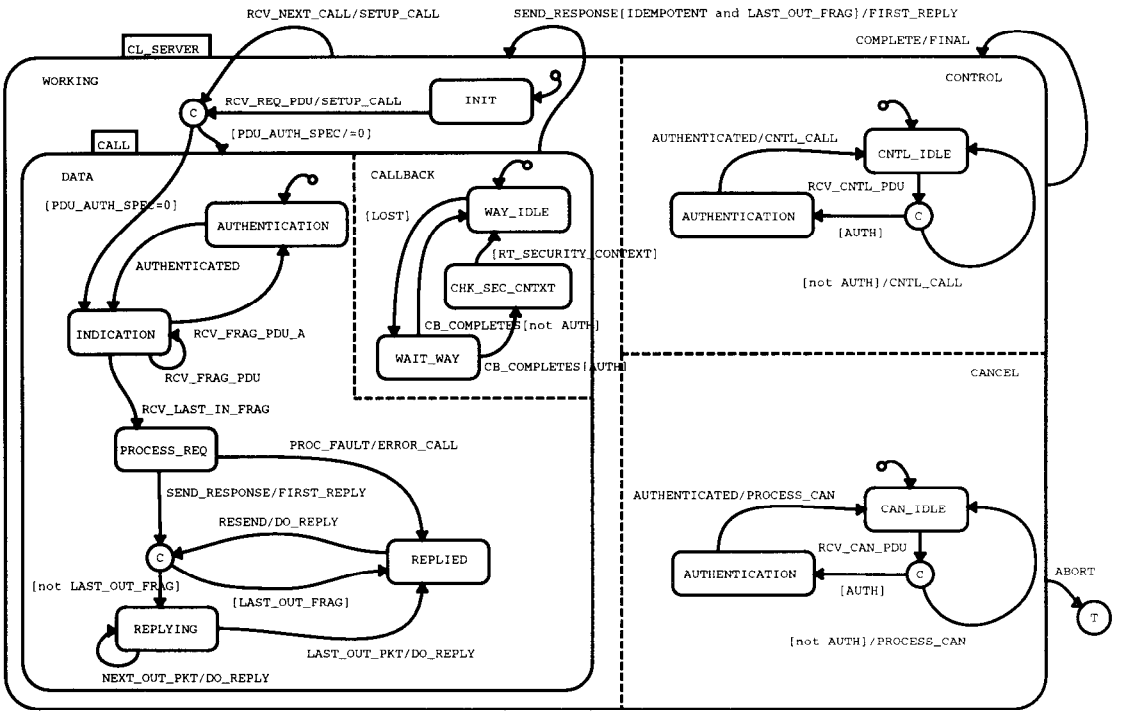


Figure 10-2 CL_SERVER Statechart

10.2.1 CL_SERVER Activities

The CL_SERVER statechart defines the following activities:

Chart: CL_SERVER

Activity: ABORT_CALL

Description: Flush and discard any further received packets for this call. If this activity was caused by a detected failure, there may be numerous additional packets in the pipeline. The flush may be lazy, upon subsequent receive processing. Also, notify the run-time system and stub to reclaim any resources for this call.

Chart: CL_SERVER

Activity: CANCEL_NOTIFY_APP

Description: This activity notifies the manager routine of the RPC application about the cancel request issued by the client.

CANCEL_NOTIFY_APP activity terminates after acknowledgement from the stub. The stub sets the RETURN_PENDING_CANCEL flag appropriately.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_SERVER

Activity: HANDLE_IN_FRAG

Description: This activity is invoked at each received fragment evaluation of **in** parameters for multi-fragmented RPC requests.

The HANDLE_IN_FRAG activity makes received data of continuous fragments available to the stub for unmarshalling and passes the object UUID (RT_OBJ_ID) to the manager routine. This does not require a transfer of control from the run-time system to the stub for each fragment; implementation policy determines when control is transferred.

In addition, if the server receives a fragment out of order, HANDLE_IN_FRAG buffers this fragment temporarily until subsequently received fragments allow for a consecutive inclusion of these temporarily buffered fragments. The fragment ordering is determined by the fragment number (RT_IN_FRAG_NUM). If previously buffered out of order fragments are appended to the continuous receive buffer (RT_IN_PARAMS), HANDLE_IN_FRAG must also adjust the state variable RT_CONT_IN_FRAG_NUM. HANDLE_IN_FRAG also maintains the selective acknowledgement bit masks which are used in the **fack** PDU.

Modifications of RT_CONT_IN_FRAG_NUM must be synchronised with other actions (DO_IN_PKT) performed by the protocol machine.

Chart: CL_SERVER

Activity: RESET_OUT_FRAG

Description: This activity is invoked if the protocol machine determines that a set of fragments needs to be retransmitted. RESET_OUT_FRAG resets the values of SND_FRAG_NUM and RT_OUT_FRAG to the beginning of the transmission queue. Note that the condition LAST_OUT_FRAG must also be set appropriately.

The first fragment in this queue may not fragment number 0, since other fragment acknowledgements may have been received, allowing the server to free previously sent data packets. Implementations must keep state about the acknowledgement of sent fragmented response PDUs.

This activity generates the event RESEND and marks TRANSMIT_RESP as true to trigger the transitions that actually invoke the appropriate SEND_PKT activities.

Chart: CL_SERVER

Activity: SEND_PKT

Description: Prepare a PDU to send to the client, adding the appropriate header information as necessary. If security services were requested (conditional flag AUTH is true), apply per-message security services. Send the PDU.

The conditional flags and data items set in the run-time system (with prefix SND_) provide the appropriate input for generating the PDU data. Note that actions within the same execution step that started this activity may have assigned values to the SND_* variables which have to be taken by this instance of the activity.

After sending a **response** PDU, the RT_OUT_FRAG pointer is incremented accordingly, to point to the remaining data in the transmit queue.

Note: The SEND_PKT activity may be invoked simultaneously by several orthogonal states (WORKING, CONTROL, CANCEL, and so on). The run-time system must catch these send requests, buffer these and the associated data, and perform the sends in sequential order.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Activity: SEND_WAY

Description: Set up and perform the conversation manager remote procedure call **conv_who_are_you**( ) (specified in Annex P). Conversation manager operations enable servers to enforce at-most-once execution semantics.

The server invokes this activity when it does not have a record of the client's call sequence number.

There are three cases in which a server will have no record of a client's sequence number:

- when the request is the first request from the client
- when the request is a duplicate and the server has executed the original request, but due to a crash, the server has not sent a response and has lost all information about the client
- when the request is a duplicate, the server has executed the original request, and the server has sent a response, but due to a delay in the client acknowledgement, the server has discarded all information about the client.

Input parameters for this call are:

h — The primitive call handle.

actuid — This is SND_ACTIVITY_ID, passed from the protocol machine; the activity UUID of the current outstanding request. In implementations that support multiple simultaneous client requests, this value is used to identify the client about whose request the server is querying.

boot_time — This is SYS_BOOT_TIME, passed from the protocol machine; the server's boot time.

Output parameters are:

seq — This is passed as PARAM_CB_SEQ_NUM to the protocol machine: the sequence number that the client associates with its current outstanding request.

st — This is passed as PARAM_CB_STATUS to the protocol machine; the status information returned by the operation. This may be one of:

CONST_RPC_S_OK	Operation succeeded.
CONST_YOU_CRASHED	The server has crashed and rebooted since establishing communication with the client.
CONST_BAD_ACT_ID	The activity UUID was wrong.

The client of the initiating RPC acts as server for an idempotent call with the same activity identifier (that is, a CL_SERVER protocol machine gets instantiated). The client (now acting as server) sets the received server boot_time (SYS_BOOT_TIME) in the client protocol machine (SND_BOOT_TIME). If the client subsequently receives a conversation manager request whose SYS_BOOT_TIME is later than the stored SND_BOOT_TIME, the client knows that the server has crashed and rebooted and sends a status CONST_YOU_CRASHED in its response.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Activity: SEND_WAY2

Description: SEND_WAY2 supersedes SEND_WAY and is called only if the server stub signals that it requires the execution context of the current call and if the run-time protocol machine has not obtained this state information yet. This activity performs the **conv_who_are_you2**( ) remote procedure call (specified in Annex P).

The additional output parameter is:

cas_uuid This is passed as PARAM_CLIENT_EXECUTION_CONTEXT to the protocol machine: a UUID that uniquely identifies the execution context (address space) of the calling client. This information is needed for servers that maintain client context state.

Note: If the RPC is an authenticated call, the run-time system should have the execution context information already, since PARAM_CLIENT_EXECUTION_CONTEXT is carried as **out** parameter of the SEND_WAYAUTH activity.

Chart: CL_SERVER

Activity: SEND_WAYAUTH

Description: SEND_WAYAUTH supersedes SEND_WAY and SEND_WAY2 and is called only for authenticated RPC requests (conditional flag AUTH is true) that do not use a previously established security context (RT_SECURITY_CONTEXT condition is false). This activity performs the **conv_who_are_you_auth**( ) remote procedure call (specified in Annex P).

The additional input parameters are:

in_len This is SND_CB_IN_LEN, passed from the protocol machine; the length in bytes of SND_CB_IN_DATA.

in_data This is SND_CB_IN_DATA, passed from the protocol machine; An array of bytes issued to the client as an authentication challenge.

Contents of in_data are determined by the authentication protocol used. Encodings for the protocol dce_c_rpc_authn_protocol_krb5 are specified in Section 13.

out_max_len This is SND_CB_OUT_MAX_LEN, passed from the protocol machine; the maximum length in bytes of the array to be returned in PARAM_CB_OUT_DATA.

The additional output parameters are:

out_len This is passed as PARAM_CB_OUT_LEN to the protocol machine; the length in bytes of PARAM_CB_OUT_DATA.

out_data This is passed as PARAM_CB_OUT_DATA to the protocol machine; an array of bytes returned to the server as an authentication response.

Contents of out_data are determined by the authentication protocol used. Encodings for dce_c_rpc_authn_protocol_krb5 protocol are specified in Section 13.

The client of the initiating RPC (now acting as server) verifies the received challenge message and sets (if succeeded) the condition flag RT_SECURITY_CONTEXT in the client protocol machine to true, otherwise it sets RT_SECURITY_CONTEXT to false and raises the event DENIED.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Activity: VERIFY_AUTH

Description: Verify the authentication trailer of PDU and decrypt message if necessary.

This activity takes as input values the PDU header field **auth_proto** (authentication protocol specifier: RT_AUTH_SPEC) and the authentication verifier (PDU trailer: RT_AUTH_VERIFIER).

Depending on the result of the verification, the activity VERIFY_AUTH generates either the event AUTHENTICATED (success) or DENIED (authentication failure).

The algorithm applied to this activity is dependent on the security service in use (determined by RT_AUTH_SPEC). The general evaluation steps for authentication service rpc_c_authn_dce_secret are as follows (for more details see Section 13):

- Check the protection level applied to the PDU (parameter in RT_AUTH_VERIFIER) against the protection level for the call (negotiated security context). If matching, proceed with verification, otherwise raise DENIED.

  Note that bind requests are used for negotiating the security context. Therefore, the protection level will not be verified for these PDUs; this verification takes only place for actual call PDUs.

- Decrypt the cyphertext portion of the verifier and verify PDUs integrity. If discrepancies are found, raise DENIED, otherwise raise AUTHENTICATED and proceed (if privacy protected).

- If privacy protection is requested, decrypt PDU body data.

Note: The VERIFY_AUTH activity may be invoked simultaneously by several orthogonal states (WORKING, CONTROL and CANCEL). VERIFY_AUTH must not generate the event AUTHENTICATED unless the entire requested authentication processing is completed. If VERIFY_AUTH detects an authentication failure and generates the event DENIED, the protocol machine rejects the RPC and no further processing is required.

Chart: CL_SERVER

Activity: VERIFY_AUTH_CONTEXT

Description: Verifies the results of the conversation manager callback (SEND_WAYAUTH) according to the authentication protocol used.

This activity evaluates the returned parameter PARAM_CB_OUT_DATA (PARAM_CB_OUT_LEN), containing the authentication response. It sets the condition flag RT_SECURITY_CONTEXT to true if verification succeeded or raises DENIED if verification failed.

10.2.2 CL_SERVER States

The CL_SERVER statechart defines the following states:

Chart: CL_SERVER

State: AUTHENTICATION

Description: Process authentication verification.

Reactions	
Trigger	**Action**
[RT_SECURITY_CONTEXT]	st!(VERIFY_AUTH)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

State: AUTHENTICATION

Description: Process authentication verification.

Reactions	
Trigger	**Action**
[RT_SECURITY_CONTEXT]	st!(VERIFY_AUTH)

Chart: CL_SERVER

State: AUTHENTICATION

Description: Process authentication verification.

Reactions	
Trigger	**Action**
[RT_SECURITY_CONTEXT]	st!(VERIFY_AUTH)

Chart: CL_SERVER

State: CALL

Description: Processing a remote procedure call request.

Reactions	
Trigger	**Action**
entering	RT_IN_PARAMS:=NULL; RT_CONT_IN_FRAG_NUM:=-1
entering	IF PDU_IDEMPOTENT THEN tr!(IDEMPOTENT) END IF
entering	IF PDU_MAYBE THEN tr!(MAYBE) END IF
entering	IF PDU_BROADCAST THEN tr!(BROADCAST) END IF
entering	IF SEQ_NUM_GT THEN RT_SEQ_NUM:=PDU_SEQ_NUM END IF

Chart: CL_SERVER

State: CALLBACK

Description: Processing of conversation manager callback procedures.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

State: CANCEL

Description: Processing of client requests to terminate the call in progress.

The reaction within this state senses the termination of the CANCEL_NOTIFY_APP activity as cancel acknowledgement from the server manager routine. The manager routine also sets the RETURN_PENDING_CANCEL flag appropriately.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Reactions	
Trigger	**Action**
entering	fs!(SND_PENDING_CANCEL); RT_CANCEL_ID:=0

sp(CANCFL_NOTIFY_APP)	IF
	  RETURN_PENDING_CANCEL
	THEN
	  tr!(SND_PENDING_CANCEL)
	END IF;
	CANACK_CALL

Chart: CL_SERVER

State: CAN_IDLE

Description: Waits for cancel requests.

Reactions	
Trigger	**Action**

exiting	IF
	  AUTH
	THEN
	  RT_AUTH_VERIFIER_CAN:=PDU_AUTH_VERIFIER
	END IF

Chart: CL_SERVER

State: CHK_SEC_CNTXT

Description: Verify the security context negotiated through SEND_WAYAUTH callback.

Activities Throughout:
VERIFY_AUTH_CONTEXT

Reactions	
Trigger	**Action**
ex(CHK_SEC_CNTXT)	
[RT_SECURITY_CONTEXT]	fs!(CONTEXT_REQUEST); RT_CLIENT_EXECUTION_CONTEXT:= PARAM_CLIENT_EXECUTION_CONTEXT

Chart: CL_SERVER

State: CL_SERVER

Description: Main state for statechart CL_SERVER.

Chart: CL_SERVER

State: CNTL_IDLE

Description: Waits for incoming control PDUs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Reactions	
Trigger	**Action**
RECEIVE_PDU[PDU_TYPE=ACK and VALID_PDU_HEADER]	tr!(ACK_PDU)
RECEIVE_PDU[PDU_TYPE=FACK and VALID_PDU_HEADER]	tr!(FACK_PDU)
RECEIVE_PDU[PDU_TYPE=PING and VALID_PDU_HEADER]	tr!(PING_PDU)
exiting	IF AUTH THEN RT_AUTH_VERIFIER_CNTL:= PDU_AUTH_VERIFIER END IF
en(CNTL_IDLE)	fs!(ACK_PDU); fs!(FACK_PDU); fs!(PING_PDU)
RECEIVE_PDU[CNTL_PDU and VALID_PDU_HEADER]	RCV_CNTL_PDU

Chart: CL_SERVER

State: CONTROL

Description: Processing received control PDUs.

Chart: CL_SERVER

State: DATA

Description: Processing the data PDUs for remote procedure call.

Reactions	
Trigger	**Action**
en(DATA)	SND_SERIAL_NUM:=0; RT_REPLY_COUNT:=0

Chart: CL_SERVER

State: INDICATION

Description: Handles incoming RPC request fragments.

Reactions	
Trigger	**Action**
en(INDICATION)[not NO_FACK or not LAST_IN_FRAG and RT_BUF_LIMIT and NO_FACK]	FACK_CALL
en(INDICATION)[not LAST_IN_FRAG and RT_IN_FRAG_NUM=RT_CONT_IN_FRAG_NUM+1]	DO_IN_PKT; st!(HANDLE_IN_FRAG)
en(INDICATION)[not LAST_IN_FRAG and RT_IN_FRAG_NUM/=RT_CONT_IN_FRAG_NUM+1]	st!(HANDLE_IN_FRAG)
RECEIVE_PDU[PDU_TYPE=REQUEST and VALID_PDU_HEADER and not AUTH]	DO_REQ; RCV_FRAG_PDU
RECEIVE_PDU[PDU_TYPE=REQUEST and VALID_PDU_HEADER and AUTH]	DO_REQ; RCV_FRAG_PDU_A
en(INDICATION)[LAST_IN_FRAG and RT_LAST_IN_FRAG_NUM= RT_CONT_IN_FRAG_NUM+1]	st!(HANDLE_IN_FRAG); RCV_LAST_IN_FRAG

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

State: INIT

Description: Initial call state. Waits for request from client.

Reactions	
Trigger	**Action**
RECEIVE_PDU[PDU_TYPE=REQUEST and not BOOT_TIME_EQ]	REJECT_CALL
tm(en(WORKING), TIMEOUT_IDLE)	st!(ABORT_CALL)

Chart: CL_SERVER

State: PROCESS_REQ

Description: Promotes completely received request to manager routine (RPC stub).

Reactions	
Trigger	**Action**
entering	RT_OUT_PARAMS:=NULL

Chart: CL_SERVER

State: REPLIED

Description: Terminal state for at-most-once calls.

Reactions	
Trigger	**Action**
tm(en(REPLIED), TIMEOUT_RESEND)	RESEND_OUT_FRAGS

Chart: CL_SERVER

State: REPLYING

Description: Handles fragmented reply to client.

Reactions	
Trigger	**Action**
tm(en(REPLYING), TIMEOUT_RESEND)	RESEND_OUT_FRAGS

Chart: CL_SERVER

State: WAIT_WAY

Description: Invoke conversation manager and wait for response.

Reactions	
Trigger	**Action**
en(WAIT_WAY)	DO_CALLBACK
ex(WAIT_WAY)[not AUTH and CONTEXT_REQUEST]	fs!(CONTEXT_REQUEST); RT_CLIENT_EXECUTION_CONTEXT:= PARAM_CLIENT_EXECUTION_CONTEXT

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_SERVER

State: WAY_IDLE

Description: Idle unless new conversation manager request.

Chart: CL_SERVER

State: WORKING

Description: Main working state for call instance.

Reactions	
Trigger	**Action**
en(WORKING)	fs!(BROADCAST); fs!(MAYBE); fs!(IDEMPOTENT); fs!(AUTH); fs!(LAST_IN_FRAG)

exiting	IF
	  PDU_AUTH_SPEC/=0
	THEN
	  tr!(AUTH)
	ELSE
	  fs!(AUTH)
	END IF

10.2.3 CL_SERVER Events

The CL_SERVER statechart defines the following events:

Chart: CL_SERVER

Event: ABORT

Description: RPC session (same activity UUID) has terminated.

Definition: `st(ABORT_CALL)`

Chart: CL_SERVER

Event: AUTHENTICATED

Description: Authentication processing completed successfully.

Chart: CL_SERVER

Event: AUTHENTICATED_RES

Description: Authentication for **cancel** PDU successful (request already processed).

Definition:
```
AUTHENTICATED[in(CANCEL.AUTHENTICATION)
 and (in(REPLYING) or in(REPLIED))]
```

Chart: CL_SERVER

Event: CB_COMPLETES

Description: Callback completes successfully: the conversation manager callback has completed.

Definition:
```
(sp(SEND_WAY) or
 sp(SEND_WAYAUTH))[PARAM_CB_STATUS=CONST_RPC_S_OK]
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Event: CB_FAULT

Description: Callback completes: client detected mismatch in sequence numbers.

Definition:
```
(sp(SEND_WAY) or
 sp(SEND_WAYAUTH))[RT_SEQ_NUM/=PARAM_CB_SEQ_NUM]
```

Chart: CL_SERVER

Event: CB_REJECT

Description: Callback completes: client detected wrong activity identifier or a server boot time error.

Definition:
```
(sp(SEND_WAY) or
 sp(SEND_WAYAUTH))[PARAM_CB_STATUS=CONST_YOU_CRASHED
 or PARAM_CB_STATUS=CONST_BAD_ACT_ID]
```

Chart: CL_SERVER

Event: COMPLETE

Description: RPC completed (with success or fault).

Definition:
```
sp(SEND_PKT)[SND_REPLY_TYPE=CANCEL_ACK]
 or CB_FAULT or CB_REJECT or
 PROCESSING_FAULT[IDEMPOTENT or BROADCAST or MAYBE]
 or COMPLETE_CLEAR or COMPLETE_FREE
 or en(REPLIED)[LAST_OUT_FRAG and IDEMPOTENT]
 or DENIED
```

Chart: CL_SERVER

Event: COMPLETE_CLEAR

Description: Ready to clear **out** parameters.

Definition:
```
PROCESSING_FDNE[IDEMPOTENT] or
 [(in(REPLYING) or in(REPLIED)) and
 RT_REPLY_COUNT>MAX_REPLIES] or
 AUTHENTICATED_RES or
 en(CNTL_IDLE)[ACK_PDU and NON_IDEMPOTENT]
```

Chart: CL_SERVER

Event: COMPLETE_FREE

Description: Ready to free activity record for requesting client.

Definition:
```
PROC_RESPONSE[MAYBE] or
 PROCESSING_FDNE[BROADCAST or MAYBE] or
 [AUTH and TICKET_EXP]
```

Chart: CL_SERVER

Event: DENIED

Description: Authentication failure detected.

The VERIFY_AUTH activity generates this event if either the integrity check failed or the requested protection level for authentication services does not match.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Event: LAST_OUT_PKT

Description: Statechart internal event: last fragment of fragmented response.

Definition:
```
[TRANSMIT_RESP and LAST_OUT_FRAG and BURST]
 or en(CNTL_IDLE)[TRANSMIT_RESP and
 FACK_PDU and LAST_OUT_FRAG and
 OUT_FRAG_NUM_EQ]
```

Chart: CL_SERVER

Event: NEXT_OUT_PKT

Description: Statechart internal event: intermediate fragment of fragmented response

Definition:
```
[TRANSMIT_RESP and not LAST_OUT_FRAG and
 BURST] or en(CNTL_IDLE)[TRANSMIT_RESP and
 FACK_PDU and not LAST_OUT_FRAG and
 OUT_FRAG_NUM_EQ]
```

Chart: CL_SERVER

Event: PROCESSING_FAULT

Description: Execution of procedure failed. Returned from called procedure (stub).

Chart: CL_SERVER

Event: PROCESSING_FDNE

Description: Stub (manager routine) or run-time system rejected RPC request.

The call did not execute.

Chart: CL_SERVER

Event: PROC_FAULT

Description: Cannot execute or fault returned from called procedure (stub).

Definition:
```
PROCESSING_FAULT or PROCESSING_FDNE[not
 BROADCAST and not MAYBE]
```

Chart: CL_SERVER

Event: PROC_RESPONSE

Description: Call returned from called procedure (server manager routine).

This event indicates that the called application procedure is ready to response to the RPC request and has provided **out** parameter data in the RT_OUT_PARAMS queue. The processing of the application procedure may not have been completed and more **out** parameter data may to be queued (sensed by the TRANSMIT_RESP and LAST_OUT_FRAG condition flags).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_SERVER

Event: RCV_CAN_PDU

Description: Received **cancel** PDU with valid header.

Definition:
```
RECEIVE_PDU[PDU_TYPE=CANCEL and
 VALID_PDU_HEADER and
 PDU_CANCEL_VERSION=CONST_CANCEL_VERSION
 and in(DATA)]
```

Chart: CL_SERVER

Event: RCV_CNTL_PDU

Description: Received one of the control PDUs (**ack**, **fack** or **ping**) with valid header.

Chart: CL_SERVER

Event: RCV_FRAG_PDU

Description: Received PDU for nonauthenticated fragmented requests with valid header.

Chart: CL_SERVER

Event: RCV_FRAG_PDU_A

Description: Received PDU for authenticated fragmented request with valid header.

Chart: CL_SERVER

Event: RCV_LAST_IN_FRAG

Description: Received last fragment of request PDU and callback completed (for at-most-once).

All fragments of a multi-fragmented request are received or a single packet request was received. RCV_LAST_IN_FRAG signals that the complete request data is available to the stub for unmarshalling, and it transfers the control from the run-time system to the stub for processing the RPC request.

Chart: CL_SERVER

Event: RCV_NEXT_CALL

Description: Receive next remote procedure call with same activity ID.

Definition:
```
RCV_REQ_PDU[in(REPLIED)]
```

Chart: CL_SERVER

Event: RCV_REQ_PDU

Description: Received **request** PDU (first packet for fragmented requests) with valid header.

Definition:
```
RECEIVE_PDU[PDU_TYPE=REQUEST and
 (PDU_FRAG and PDU_FRAG_NUM=0 or not
 PDU_FRAG) and SEQ_NUM_GT and BOOT_TIME_EQ
 and PDU_VERSION_NUM=CL_VERSION_NUM_V20]
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Event: RECEIVE_PDU

Description: Received a PDU from client.

Chart: CL_SERVER

Event: RESEND

Description: Statechart internal event that triggers a resend of complete reply PDUs.

Chart: CL_SERVER

Event: SEND_RESPONSE

Description: Called procedure provided **out** parameters to be sent.

Definition:
```
PROC_RESPONSE[not MAYBE]
```

10.2.4 CL_SERVER Actions

The CL_SERVER statechart defines the following actions:

Chart: CL_SERVER

Action: CANACK_CALL

Description: Set up **cancel_ack** PDU to be sent.

The body data of this cancel acknowledgement message consists of:

- CONST_CANCEL_VERSION (that is, version number 0)
- RT_CANCEL_ID
- RT_PENDING_CANCEL.

(See also the PDU encoding of cancel_ack.)

Definition:
```
SND_REPLY_TYPE:=CANCEL_ACK;
st!(SEND_PKT)
```

Chart: CL_SERVER

Action: CNTL_CALL

Description: Reactions on received control PDUs.

Definition:
```
IF
  PING_PDU
THEN
  PING_CALL
END IF;
IF
  FACK_PDU
THEN
  EVAL_FACK_BODY;
  IF
     in(REPLYING) and OUT_FRAG_NUM_NE or
      in(REPLIED) and SEQ_NUM_LE
  THEN
     RESEND_OUT_FRAGS
  END IF
END IF
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Action: DO_CALLBACK

Description: Initialise and start activity SEND_WAY (conversation manager callback procedure).

Definition:

```
SND_ACTIVITY_ID:=RT_ACTIVITY_ID;
SND_AUTH_SPEC:=RT_AUTH_SPEC;
IF
  not AUTH and not CONTEXT_REQUEST or AUTH
   and RT_SECURITY_CONTEXT
THEN
  st!(SEND_WAY)
END IF;
IF
  not AUTH and CONTEXT_REQUEST
THEN
  st!(SEND_WAY2)
END IF;
IF
  AUTH and not RT_SECURITY_CONTEXT
THEN
  st!(SEND_WAYAUTH)
END IF
```

Chart: CL_SERVER

Action: DO_IN_PKT

Description: Append received **request** PDU body data to internal buffer.

Definition:

```
RT_CONT_IN_FRAG_NUM:=RT_CONT_IN_FRAG_NUM+1;
RT_IN_PARAMS:=RT_IN_PARAMS+RT_BODY
```

Chart: CL_SERVER

Action: DO_REPLY

Description: Send **out** fragment to requesting client.

Definition:

```
fs!(TRANSMIT_RESP);
IF
  LAST_OUT_FRAG
THEN
  tr!(SND_LAST_FRAG)
ELSE
  fs!(SND_LAST_FRAG)
END IF;
SND_FRAG_NUM:=SND_FRAG_NUM+1;
SND_SERIAL_NUM:=SND_SERIAL_NUM+1;
SND_OUT_PARAMS:=RT_OUT_FRAG;
SND_REPLY_TYPE:=RESPONSE;
st!(SEND_PKT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Action: DO_REQ

Description: Evaluate **request** PDU header.

Definition:

```
RT_BODY:=PDU_BODY;
RT_IN_FRAG_NUM:=PDU_FRAG_NUM;
RT_IN_SERIAL_NUM:=PDU_SERIAL_NUM;
IF
  PDU_AUTH_SPEC/=0
THEN
  RT_AUTH_VERIFIER_CALL:=PDU_AUTH_VERIFIER
END IF;
IF
  PDU_NO_FACK or PDU_FRAG
THEN
  tr!(NO_FACK)
ELSE
  fs!(NO_FACK)
END IF;
IF
  PDU_LAST_FRAG or not PDU_FRAG
THEN
  tr!(LAST_IN_FRAG);
  RT_LAST_IN_FRAG_NUM:=PDU_FRAG_NUM
END IF
```

Chart: CL_SERVER

Action: ERROR_CALL

Description: Set up error PDU (**fault** or **reject**) to be sent.

Definition:

```
WHEN
  PROCESSING_FAULT[not BROADCAST and not
   MAYBE]
THEN
  RT_OUT_PARAMS:=SND_FAULT_STATUS;
  SND_REPLY_TYPE:=FAULT;
  st!(SEND_PKT)
ELSE
  WHEN
     PROCESSING_FDNE[not BROADCAST and
      not MAYBE]
  THEN
     RT_OUT_PARAMS:=SND_REJECT_STATUS;
     SND_REPLY_TYPE:=REJECT;
     st!(SEND_PKT)
  END WHEN
END WHEN
```

Chart: CL_SERVER

Action: EVAL_FACK_BODY

Description: Invoke implementation-specific activity to evaluate **fack** body data.

This action reads the **fack** PDU body data according to the PDU specification. It is RPC run-time implementation-specific how this data will be evaluated and used for subsequent fragmented transmissions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Definition:
```
rd!(PDU_FACK_BODY)
```

Chart: CL_SERVER

Action: FACK_CALL

Description: Send **fack** PDU if **nofack** flag is false or receiver has buffer full condition.

Definition:
```
SND_OUT_PARAMS:=RT_FACK_BC
SND_REPLY_TYPE:=FACK;
st!(SEND_PKT)
```

Chart: CL_SERVER

Action: FINAL

Description: Perform final actions for RPC.

Definition:
```
WHEN
  COMPLETE_CLEAR
THEN
  RT_OUT_PARAMS:=NULL
END WHEN;
WHEN
  COMPLETE_FREE
THEN
  st!(ABORT_CALL)
END WHEN;
WHEN
  DENIED
THEN
  SND_OUT_PARAMS:=CONST_NCA_S_INVALID_CHKSUM;
  SND_REPLY_TYPE:=REJECT;
  st!(SEND_PKT)
END WHEN;
WHEN
  CB_REJECT
THEN
  SND_OUT_PARAMS:=PARAM_CB_STATUS;
  SND_REPLY_TYPE:=REJECT;
  st!(SEND_PKT)
END WHEN
```

Chart: CL_SERVER

Action: FIRST_REPLY

Description: Initialise and send first reply PDU.

Definition:
```
fs!(TRANSMIT_RESP);
IF
  LAST_OUT_FRAG
THEN
  fs!(SND_FRAG);
  tr!(SND_LAST_FRAG)
ELSE
  tr!(SND_FRAG);
  fs!(SND_LAST_FRAG)
END IF;
SND_SEQ_NUM:=RT_SEQ_NUM;
SND_IF_ID:=RT_IF_ID;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
SND_IF_VERSION:=RT_IF_VERSION;
SND_OBJ_ID:=RT_OBJ_ID;
SND_OP_NUM:=RT_OP_NUM;
SND_ACTIVITY_ID:=RT_ACTIVITY_ID;
SND_AUTH_SPEC:=RT_AUTH_SPEC;
SND_BOOT_TIME:=RT_BOOT_TIME;
RT_OUT_FRAG:=RT_OUT_PARAMS;
SND_OUT_PARAMS:=RT_OUT_PARAMS;
SND_FRAG_NUM:=0;
SND_REPLY_TYPE:=RESPONSE;
st!(SEND_PKT)
```

Chart: CL_SERVER

Action: NO_CALL

Description: Set up nocall PDU to be sent.

Definition:

```
SND_OUT_PARAMS:=RT_FACK_BODY;
SND_REPLY_TYPE:=NOCALL;
st!(SEND_PKT)
```

Chart: CL_SERVER

Action: PING_CALL

Description: Actions in response to a ping PDU call.

Definition:

```
IF
  BOOT_TIME_EQ
THEN
  IF
     in(INIT) or (in(INDICATION) or
      in(DATA.AUTHENTICATION)) and
      SEQ_NUM_EQ or SEQ_NUM_GT
  THEN
     NO_CALL
  END IF;
  IF
     in(PROCESS_REQ) and SEQ_NUM_EQ
  THEN
     SND_REPLY_TYPE:=WORKING;
     st!(SEND_PKT)
  END IF;
  IF
     (in(REPLYING) or in(REPLIED)) and
      SEQ_NUM_LE
  THEN
     RESEND_OUT_FRAGS
  END IF
ELSE
  SND_OUT_PARAMS:=CONST_WRONG_BOOT_TIME;
  SND_REPLY_TYPE:=REJECT;
  st!(SEND_PKT)
END IF
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Action: PROCESS_CAN

Description: Process cancel request (signal manager routine).

Definition:

```
IF
  PDU_CANCEL_ID>RT_CANCEL_ID
THEN
  RT_CANCEL_ID:=PDU_CANCEL_ID
END IF;
IF
  in(DATA) and not in(REPLYING) and not
   in(REPLIED)
THEN
  st!(CANCEL_NOTIFY_APP)
END IF;
IF
  in(REPLYING) or in(REPLIED)
THEN
  tr!(SND_PENDING_CANCEL);
  CANACK_CALL
END IF
```

Chart: CL_SERVER

Action: REJECT_CALL

Description: Perform a reject call.

Definition:

```
IF
  not BOOT_TIME_EQ
THEN
  SND_OUT_PARAMS:=CONST_WRONG_BOOT_TIME
ELSE
  SND_OUT_PARAMS:=CONST_UNSPEC_REJECT
END IF;
SND_REPLY_TYPE:=REJECT;
st!(SEND_PKT)
```

Chart: CL_SERVER

Action: RESEND_OUT_FRAGS

Description: Perform a resend of previously sent **response** PDUs.

Definition:

```
fs!(TRANSMIT_RESP);
RT_REPLY_COUNT:=RT_REPLY_COUNT+1;
IF
  SND_FRAG
THEN
  fs!(LAST_OUT_FRAG)
END IF;
st!(RESET_OUT_FRAG)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Action: SETUP_CALL

Description: Set up call data at first call's **request** PDU.

Definition:

```
RT_CLIENT_EXECUTION_CONTEXT:=NULL;
fs!(CONTEXT_REQUEST);
tr!(NO_FACK);
RT_ACTIVITY_ID:=PDU_ACTIVITY_ID;
RT_SEQ_NUM:=PDU_SEQ_NUM;
RT_IF_ID:=PDU_IF_ID;
RT_IF_VERSION:=PDU_IF_VERSION;
RT_OBJ_ID:=PDU_OBJ_ID;
RT_OP_NUM:=PDU_OP_NUM;
RT_AUTH_SPEC:=PDU_AUTH_SPEC;
IF
  PDU_AUTH_SPEC/=0
THEN
  RT_AUTH_VERIFIER_CALL:=PDU_AUTH_VERIFIER
END IF
```

10.2.5 CL_SERVER Conditions

The CL_SERVER statechart defines the following conditions:

Chart: CL_SERVER

Condition: ACK_PDU

Description: Statechart internal flag: received PDU type **ack**.

Chart: CL_SERVER

Condition: AUTH

Description: Statechart internal flag: false if PDU **auth_id** = 0; true otherwise.

Chart: CL_SERVER

Condition: BOOT_TIME_EQ

Description: Statechart internal flag.

Definition:

```
PDU_BOOT_TIME=SYS_BOOT_TIME or
 PDU_BOOT_TIME=0 and (RT_IN_FRAG_NUM=0 or
 not PDU_FRAG)
```

Chart: CL_SERVER

Condition: BROADCAST

Description: Statechart internal flag: broadcast call semantic.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Condition: BURST

Description: Run time internal flag set if no **fack** is expected before sending next fragment.

This flag is used by run-time implementations to optimise the frequency of fragmented outbound packets.

The algorithms used to optimise traffic and avoid congestion are implementation-specific. The protocol machine (state RESP_WAIT) waits for inbound **fack** PDUs if burst mode is off. The next outbound fragment is triggered by an inbound **fack** PDU. Implementations are responsible for setting the corresponding **nofack** flags in the PDU header.

Chart: CL_SERVER

Condition: CNTL_PDU

Description: Statechart internal flag: control PDUs to be received.

Definition:
```
PDU_TYPE=ACK or PDU_TYPE=PING or
 PDU_TYPE=FACK
```

Chart: CL_SERVER

Condition: CONTEXT_REQUEST

Description: Stub requests client's execution context for which it has no record.

Chart: CL_SERVER

Condition: FACK_PDU

Description: Statechart internal flag: received PDU type **fack**.

Chart: CL_SERVER

Condition: IDEMPOTENT

Description: Statechart internal flag: idempotent call.

Chart: CL_SERVER

Condition: LAST_IN_FRAG

Description: Statechart internal flag: last **in** fragment or non-frag **in** packet received.

Chart: CL_SERVER

Condition: LAST_OUT_FRAG

Description: Statechart internal flag: last **out** fragment or non-frag **out** packet ready to send.

This flag is set by the run-time system if the transmit queue contains the last fragment (see also Subclause 9.3 on page 304).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Condition: LOST

Description: Statechart internal: server boot time = 0 or client's context or cached sequence number lost.

Definition:
```
(PDU_BOOT_TIME=0 or SEQ_NUM_LOST) and
 NON_IDEMPOTENT or AUTH and not
 RT_SECURITY_CONTEXT or CONTEXT_REQUEST and
 RT_CLIENT_EXECUTION_CONTEXT=NULL
```

Chart: CL_SERVER

Condition: MAYBE

Description: Statechart internal flag: maybe call.

Chart: CL_SERVER

Condition: NON_IDEMPOTENT

Description: Statechart internal flag: non-idempotent (at-most-once) call.

Definition:
```
not IDEMPOTENT and not BROADCAST and not MAYBE
```

Chart: CL_SERVER

Condition: NO_FACK

Description: Statechart internal flag: received PDU with **nofack** flag true.

Chart: CL_SERVER

Condition: OUT_FRAG_NUM_EQ

Description: Statechart internal flag: received fragment number at client and last sent fragment number are equal.

This condition verifies the fragment number that was received in a **fack** PDU. (See Section 12 for details.)

Definition:
```
SND_FRAG_NUM=PDU_FRAG_NUM
```

Chart: CL_SERVER

Condition: OUT_FRAG_NUM_NE

Description: Statechart internal flag: received fragment number at client and last sent fragment number are not equal.

This condition verifies the fragment number that was received in a **fack** PDU. (See Section 12 for details.)

Definition:
```
SND_FRAG_NUM/=PDU_FRAG_NUM
```

Chart: CL_SERVER

Condition: PDU_BROADCAST

Description: PDU flag **broadcast**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Condition: PDU_FRAG
Description: PDU flag **frag**.

Chart: CL_SERVER
Condition: PDU_IDEMPOTENT
Description: PDU flag **idempotent**.

Chart: CL_SERVER
Condition: PDU_LAST_FRAG
Description: PDU flag **lastfrag**.

Chart: CL_SERVER
Condition: PDU_MAYBE
Description: PDU flag **maybe**.

Chart: CL_SERVER
Condition: PDU_NO_FACK
Description: PDU flag nofack.

Chart: **CL_SERVER**
Condition: PING_PDU
Description: Statechart internal flag: received PDU type **ping**.

Chart: CL_SERVER
Condition: RETURN_PENDING_CANCEL
Description: Cancel pending state returned from stub after processing the cancel request.

Chart: CL_SERVER
Condition: RT_BUF_LIMIT
Description: Statechart internal flag: buffer limit reached for **in** packets.

The conditional flag RT_BUF_LIMIT triggers the generation of a **fack** PDU which requests the sender of data fragments to readjust the transmission rate.

This is a mechanism to indicate internal buffer limits (overflow) for avoidance of congestion and retransmissions. Since recipients may evaluate **fack** body data in an implementation-dependent way, implementations must not rely on changes in the transmission rate. This indication is an advisory.

Run-time implementations are responsible for setting the RT_BUF_LIMIT flag according to their own policies.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Condition: RT_SECURITY_CONTEXT
Description: Security context associated with call activity UUID is set up and valid.

Chart: CL_SERVER
Condition: SEQ_NUM_EQ
Description: Statechart internal flag: received sequence number equals cached sequence number.
Definition: `PDU_SEQ_NUM=RT_SEQ_NUM`

Chart: CL_SERVER
Condition: SEQ_NUM_GT
Description: Statechart internal flag: received sequence number > cached sequence number.
Definition: `PDU_SEQ_NUM>RT_SEQ_NUM`

Chart: CL_SERVER
Condition: SEQ_NUM_LE
Description: Statechart internal flag: received sequence number ≤ cached sequence number.
Definition: `PDU_SEQ_NUM<=RT_SEQ_NUM`

Chart: CL_SERVER
Condition: SEQ_NUM_LOST
Description: Statechart internal flag: cached sequence number invalid.
Definition: `RT_SEQ_NUM=0`

Chart: CL_SERVER
Condition: SND_FRAG
Description: Statechart internal flag: header flag **frag** of fragments to be sent.

Chart: CL_SERVER
Condition: SND_LAST_FRAG
Description: Statechart internal flag: header flag **lastfrag** for PDU to be sent.

Chart: CL_SERVER
Condition: SND_PENDING_CANCEL
Description: Cancel pending state for current call at server.

Chart: CL_SERVER
Condition: TICKET_EXP
Description: Statechart internal flag: ticket expired.
Definition: `SYS_TIME>GRACE_PERIOD+PDU_EXP_TIME`

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER

Condition: TRANSMIT_RESP

Description: One or more fragments queued for transmission of response data.

This flag indicates that one or more response fragments are queued in a run-time internal buffer and ready to be transmitted. In conjunction with the BURST flag and possibly expected **fack** PDUs, an event for transmitting the next fragment will be generated.

The run-time system internally sets this flag (true) after the stub initially provides data in the transmit queue, sufficient for at least the first PDU fragment to be transmitted. The protocol machine resets this flag if it has detected and taken an event for sending the next fragment in the queue.

The run-time system sets this flag again after completion of a SEND_PKT activity if the transmit queue contains enough data for the next PDU fragment to be transmitted.

Chart: CL_SERVER

Condition: VALID_PDU_HEADER

Description: Pre-evaluation of PDU header (before authentication processing).

Definition:
```
PDU_ACTIVITY_ID=RT_ACTIVITY_ID and
 PDU_AUTH_SPEC=RT_AUTH_SPEC and SEQ_NUM_EQ
 and BOOT_TIME_EQ and
 PDU_VERSION_NUM=CL_VERSION_NUM_V20
```

10.2.6 CL_SERVER Data-Items

The CL_SERVER statechart defines the following data items:

Chart: CL_SERVER

Data Item: ACK

Description: Constant: PDU type **ack**.

Definition: 7

Chart: CL_SERVER

Data Item: CANCEL

Description: Constant: PDU type **cancel**.

Definition: 8

Chart: CL_SERVER

Data Item: CANCEL_ACK

Description: Constant: PDU **cancel_ack**.

Definition: 10

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: CL_VERSION_NUM_V20
Description: Constant: RPC protocol version 2.0 version number.
Definition: 4

Chart: CL_SERVER
Data Item: CONST_BAD_ACT_ID
Description: Reject status code for WAY callback. The encoding is specified in Annex P.

Chart: CL_SERVER
Data Item: CONST_CANCEL_VERSION
Description: Supported version number for **cancel** PDU body data.
Definition: 0

Chart: CL_SERVER
Data Item: CONST_NCA_S_INVALID_CHKSUM
Description: Constant: reject status code **nca_s_invalid_chksum**.

Chart: CL_SERVER
Data Item: CONST_RPC_S_OK
Description: Constant: status code for successful completion of WAY callback.
Definition: 0

Chart: CL_SERVER
Data Item: CONST_UNSPEC_REJECT
Description: Constant: unspecified reject status code (0x1C000009).

Chart: CL_SERVER
Data Item: CONST_WRONG_BOOT_TIME
Description: Reject status code.

Chart: CL_SERVER
Data Item: CONST_YOU_CRASHED
Description: Reject status code for WAY callback. Specified in Annex P.

Chart: CL_SERVER
Data Item: FACK
Description: Constant: PDU type **fack**.
Definition: 9

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: FAULT
Description: Constant: PDU type **fault**.
Definition: 3

Chart: CL_SERVER
Data Item: GRACE_PERIOD
Description: Grace period on server after ticket expiration (implementation-specific).

Chart: CL_SERVER
Data Item: MAX_REPLIES
Description: Maximum number of times a **response** PDU should be resent.

Chart: CL_SERVER
Data Item: NOCALL
Description: Constant: PDU type **nocall**.
Definition: 5

Chart: CL_SERVER
Data Item: PARAM_CB_OUT_DATA
Description: Callback **out** parameter: the returned authentication response as array of bytes.

Chart: CL_SERVER
Data Item: PARAM_CB_OUT_LEN
Description: Callback **out** parameter: length of received PARAM_CB_OUT_DATA field.

Chart: CL_SERVER
Data Item: PARAM_CB_SEQ_NUM
Description: Callback **out** parameter: sequence number.

Chart: CL_SERVER
Data Item: PARAM_CB_STATUS
Description: Callback **out** parameter: status.

Chart: CL_SERVER
Data Item: PARAM_CLIENT_EXECUTION_CONTEXT
Description: Callback **out** parameter: client address space UUID (execution context).

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CL_SERVER
Data Item: PDU_ACTIVITY_ID
Description: PDU header field: **act_id**.

Chart: CL_SERVER
Data Item: PDU_AUTH_SPEC
Description: PDU header field: **auth_proto**.

Chart: CL_SERVER
Data Item: PDU_AUTH_VERIFIER
Description: PDU trailer: authentication verifier (authentication protocol-specific).

Chart: CL_SERVER
Data Item: PDU_BODY
Description: Array of PDU body data.

Chart: CL_SERVER
Data Item: PDU_BOOT_TIME
Description: PDU header field: **server_boot**. Zero at first request from client.

Chart: CL_SERVER
Data Item: PDU_CANCEL_ID
Description: **cancel_id** of received **cancel** PDU body data.

Chart: **CL_SERVER**
Data Item: PDU_CANCEL_VERSION
Description: Version number (vers) of cancel PDUs body data format (currently supported version 0).

Chart: CL_SERVER
Data Item: PDU_EXP_TIME
Description: Ticket expiration time transmitted in the authentication verifier.

Chart: CL_SERVER
Data Item: PDU_FACK_BODY
Description: Body information of **fack** PDU (implementation-dependent).

Chart: CL_SERVER
Data Item: PDU_FRAG_NUM
Description: PDU header field: **frag_num**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: PDU_IF_ID
Description: PDU header field: **if_id**.

Chart: CL_SERVER
Data Item: PDU_IF_VERSION
Description: PDU header field: **if_vers**.

Chart: CL_SERVER
Data Item: PDU_OBJ_ID
Description: PDU header field: **object**.

Chart: CL_SERVER
Data Item: PDU_OP_NUM
Description: PDU header field: **opnum**.

Chart: CL_SERVER
Data Item: PDU_SEQ_NUM
Description: PDU header field: **seqnum**.

Chart: CL_SERVER
Data Item: PDU_SERIAL_NUM
Description: PDU header field: **serial_hi**.

Chart: CL_SERVER
Data Item: PDU_TYPE
Description: PDU header field: **ptype**.

Chart: CL_SERVER
Data Item: PDU_VERSION_NUM
Description: PDU header field: **rpc_vers**.

Chart: CL_SERVER
Data Item: PING
Description: Constant: PDU type **ping**.
Definition: 1

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: REJECT
Description: Constant: PDU type **reject**.
Definition: 6

Chart: CL_SERVER
Data Item: REQUEST
Description: Constant: PDU type **request**.
Definition: 0

Chart: CL_SERVER
Data Item: RESPONSE
Description: Constant: PDU type **response**.
Definition: 2

Chart: CL_SERVER
Data Item: RT_ACTIVITY_ID
Description: Statechart internal: activity UUID of current RPC.

Chart: CL_SERVER
Data Item: RT_AUTH_SPEC
Description: Statechart internal: authentication protocol specifier received and used in call.

Chart: CL_SERVER
Data Item: RT_AUTH_VERIFIER_CALL
Description: Statechart internal: authentication verifier received in CALL state.

Chart: CL_SERVER
Data Item: RT_AUTH_VERIFIER_CAN
Description: Authentication verifier received for **cancel** PDU.

Chart: CL_SERVER
Data Item: RT_AUTH_VERIFIER_CNTL
Description: Authentication verifier received for control PDU.

Chart: CL_SERVER
Data Item: RT_BODY
Description: Statechart internal: temporarily buffered **request** PDU body data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: RT_BOOT_TIME
Description: Statechart internal: boot time of server system.

Chart: CL_SERVER
Data Item: RT_CANCEL_ID
Description: Statechart internal: identifier for received cancel request with highest count.

Chart: CL_SERVER
Data Item: RT_CLIENT_EXECUTION_CONTEXT
Description: The UUID uniquely identifying the execution context (address space) of the client.

Chart: CL_SERVER
Data Item: RT_CONT_IN_FRAG_NUM
Description: Statechart internal: last fragment number of continuously buffered **in** block.

Chart: CL_SERVER
Data Item: RT_FACK_BODY
Description: Statechart internal: body data for fack PDU.

The run-time implementation must ensure that fack and **nocall** PDU body data is generated in accordance with the specifications given in Section 12.

Chart: CL_SERVER
Data Item: RT_IF_ID
Description: Statechart internal: buffered interface UUID of RPC.

Chart: CL_SERVER
Data Item: RT_IF_VERSION
Description: Statechart internal: buffered interface version of RPC.

Chart: CL_SERVER
Data Item: RT_IN_FRAG_NUM
Description: Statechart internal: fragment number of currently received **request** PDU.

Chart: CL_SERVER
Data Item: RT_IN_PARAMS
Description: Statechart internal: buffered array of reassembled input data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: **CL_SERVER**

Data Item: **RT_IN_SERIAL_NUM**

Description: Statechart internal: serial number of previously received fragment.

Chart: CL_SERVER

Data Item: RT_LAST_IN_FRAG_NUM

Description: Fragment number of last **in** fragment of remote procedure call.

Chart: CL_SERVER

Data Item: RT_OBJ_ID

Description: Statechart internal: buffered object UUID of RPC.

Chart: **CL_SERVER**

Data Item: RT_OP_NUM

Description: Statechart internal: buffered operation number of RPC.

Chart: CL_SERVER

Data Item: RT_OUT_FRAG

Description: Statechart internal pointer to data to be sent in next **response** PDU.

The SEND_PKT activity increments this pointer after a response PDU was sent.

Chart: CL_SERVER

Data Item: RT_OUT_PARAMS

Description: Buffered array of unfragmented output data.

RT_OUT_PARAMS is the queue of transmit data provided by the stub. A possible segmentation of this queue is not equivalent to the sizes of PDU fragments sent by the runtime system (SEND_PKT) activity.

The RT_OUT_FRAG variable is a pointer data type that points to the to be transmitted data fragment within this RT_IN_PARAMS queue.

Chart: CL_SERVER

Data Item: RT_REPLY_COUNT

Description: Counter for transmitted replies.

Chart: CL_SERVER

Data Item: RT_SEQ_NUM

Description: Sequence number of previously received PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: SND_ACTIVITY_ID
Description: Activity UUID to be sent.

Chart: CL_SERVER
Data Item: SND_AUTH_SPEC
Description: Authentication specifier to be sent.

Chart: CL_SERVER
Data Item: SND_BOOT_TIME
Description: Boot time to be sent.

Chart: CL_SERVER
Data Item: SND_CB_IN_DATA
Description: Callback **in** parameter: the authentication challenge as an array of bytes.

Chart: CL_SERVER
Data Item: SND_CB_IN_LEN
Description: Callback **in** parameter: the length of SND_CB_IN_DATA parameter.

Chart: CL_SERVER
Data Item: SND_CB_OUT_MAX_LEN
Description: Callback **in** parameter: the maximum length for the **out** field PARAM_CB_OUT_DATA.

Chart: CL_SERVER
Data Item: SND_FAULT_STATUS
Description: Fault status associated with the fault PDU body represented as NDR unsigned long.

Chart: CL_SERVER
Data Item: SND_FRAG_NUM
Description: Fragment number of PDU to be sent.

Chart: CL_SERVER
Data Item: SND_IF_ID
Description: Interface UUID to be sent.

Chart: CL_SERVER
Data Item: SND_IF_VERSION
Description: Interface version number to be sent.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: SND_OBJ_ID
Description: Object UUID to be sent.

Chart: CL_SERVER
Data Item: SND_OP_NUM
Description: Operation number to be sent.

Chart: CL_SERVER
Data Item: SND_OUT_PARAMS
Description: PDU body data promoted to SEND_PKT activity.

Chart: CL_SERVER
Data Item: SND_REJECT_STATUS
Description: Reject status code associated with **reject** PDU body represented as NDR type.

Chart: CL_SERVER
Data Item: SND_REPLY_TYPE
Description: PDU type to be sent.

Chart: CL_SERVER
Data Item: SND_SEQ_NUM
Description: Sequence number of PDU to be sent.

Chart: CL_SERVER
Data Item: SND_SERIAL_NUM
Description: Serial number of PDU to be sent.

Chart: CL_SERVER
Data Item: SYS_BOOT_TIME
Description: Boot time of server system: an implementation-specific value.

Chart: CL_SERVER
Data Item: SYS_TIME
Description: Secure reference time of local system

Chart: CL_SERVER
Data Item: TIMEOUT_IDLE
Description: Timeout value for keeping client's activity record.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CL_SERVER
Data Item: TIMEOUT_RESEND
Description: Timeout value for retransmitting a **response** PDU.

Chart: CL_SERVER
Data Item: WORKING
Description: Constant: PDU type **working**.
Definition: 4

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 11. Connection-oriented RPC Protocol Machines

This Section specifies the connection-oriented RPC protocol as a series of statecharts and accompanying tables of definitions.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.1 CO_CLIENT Machine

Figure 11-1 shows the CO_CLIENT machine statechart.

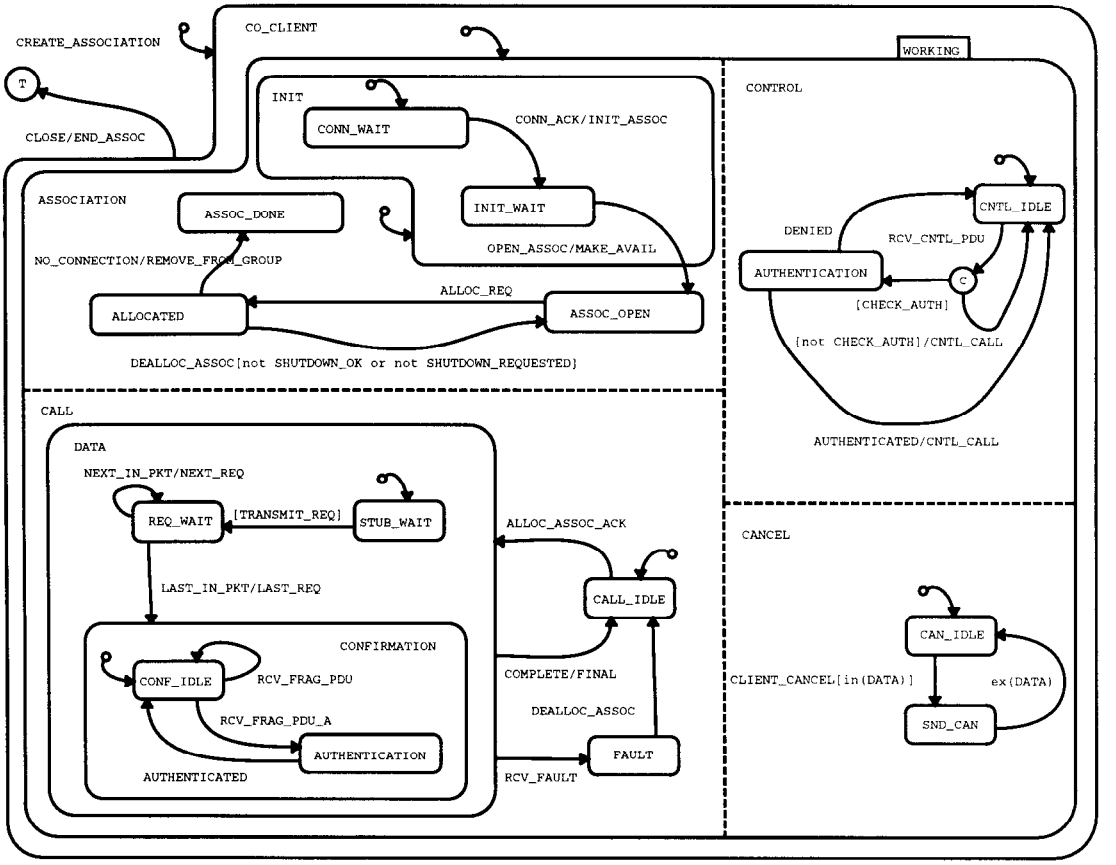


Figure 11-1 CO_CLIENT Statechart

11.1.1 CO_CLIENT Activities

The CO_CLIENT statechart defines the following activities:

Chart: CO_CLIENT

Activity: ABORT_RECEIVE

Description: Flush and discard any further responses for this call. There may be numerous additional packets in the pipeline. The flush may be lazy, upon subsequent receive processing. Also, notify the run-time system and stub to reclaim any resources for this call.

Chart: CO_CLIENT

Activity: ABORT_SEND

Description: Discontinue any further transmission of request data for the current call, to the best extent possible. Some error condition has caused a fault.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT

Activity: **DEQUEUE_CANCEL**

Description: If there is a pending cancel timeout for this call, dequeue it.

Chart: CO_CLIENT

Activity: EXCEPTION

Description: Raise a fault and return to calling routine.

Chart: CO_CLIENT

Activity: HANDLE_OUT_FRAG

Description: This activity is invoked at each received fragment evaluation of **out** parameters for multi-fragmented RPC responses.

The HANDLE_OUT_FRAG activity makes received data of the next fragment available to the stub for unmarshaling and the object UUID (RT_OBJ_ID) available to the manager routine. This does not require a transfer of control from the run-time system to the stub for each fragment; implementation policy determines when control is transferred.

Chart: CO_CLIENT

Activity: SEND_PKT

Description: Prepare a PDU to send to the server, adding the appropriate header information as necessary. If security services were requested (conditional flag AUTH is true), apply per-message security services. Send the PDU.

The conditional flags and data items set in the run-time system (with prefix SND_) provide the appropriate input for generating the PDU data. Note that actions within the same execution step that started this activity may have assigned values to the SND_* variables which have to be taken by this instance of the activity.

After sending a **request** PDU, the RT_IN_FRAG pointer is incremented accordingly, to point to the remaining data in the transmit queue.

Note: The SEND_PKT activity may be invoked simultaneously by several orthogonal states (DATA, CONTROL, CANCEL, and so on). The run-time system must catch these send requests, buffer these and the associated data, and perform the sends in sequential order.

Chart: CO_CLIENT

Activity: SET_PRES_CONTEXT

Description: Set the negotiated presentation context (from RT_SERVER_PRES_CONTEXT_LIST) and save the security credentials for this call.

This activity sets the conditional flag PRES_SEC_CONTEXT_SUPPORTED true on success, otherwise false. The selected presentation context identifier is assigned to the data item RT_PRES_CONTEXT_ID for use in subsequent messages.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Activity: VERIFY_AUTH

Description: Verify the authentication trailer of PDU and decrypt message if necessary.

This activity takes as input values the PDU header field **auth_proto** (RT_AUTH_SPEC) and the authentication verifier (RT_AUTH_VERIFIER).

Depending on the result of the verification, the activity VERIFY_AUTH generates either the event AUTHENTICATED (success) or DENIED (authentication failure).

The algorithm applied to this activity is dependent on the security service in use (determined by RT_AUTH_SPEC). The general evaluation steps for authentication service rpc_c_authn_dce_secret are as follows (for more details see Section 13):

- Check the protection level applied to the PDU (parameter in RT_AUTH_VERIFIER) against the protection level for the call (negotiated security context). If matching, proceed with verification, otherwise raise DENIED.
- Decrypt the cyphertext portion of the verifier and verify the PDU's integrity. If discrepancies are found, raise DENIED, otherwise raise AUTHENTICATED and proceed (if privacy protected).
- If privacy protection is requested, decrypt PDU body data.

Note: The VERIFY_AUTH activity may be invoked simultaneously by several orthogonal states (DATA, CONTROL and CANCEL). VERIFY_AUTH must not generate the event AUTHENTICATED unless the entire requested authentication processing is completed. If VERIFY_AUTH detects an authentication failure and generates the event DENIED, the protocol machine rejects the RPC call and no further processing is required.

11.1.2 CO_CLIENT States

The CO_CLIENT statechart defines the following states:

Chart: CO_CLIENT

State: ALLOCATED

Description: An association is allocated for current call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Reactions	
Trigger	**Action**
ALTER_CONTEXT_REQ	SND_PRES_CONTEXT_LIST:= DESIRED_CONTEXT_LIST; SND_REQUEST_TYPE:=ALTER_CONTEXT; st!(SEND_PKT)

en(ALLOCATED)	IF
	PRES_SEC_CONTEXT_SUPPORTED
	THEN
	ALLOC_ASSOC_ACK
	ELSE
	ALTER_CONTEXT_REQ
	END IF

CHECK_CONTEXT	IF
	PRES_SEC_CONTEXT_SUPPORTED
	THEN
	ALLOC_ASSOC_ACK
	ELSE
	ALLOC_ASSOC_NAK;
	DEALLOC_ASSOC
	END IF

RCV_SHUTDOWN or RESOURCES_SCARCE	tr!(SHUTDOWN_REQUESTED)
RCV_ALTER_CONTEXT_RESP	MARK_ASSOC; CHECK_CONTEXT

Chart: CO_CLIENT

State: ASSOCIATION

Description: Main state for an association.

Chart: CO_CLIENT

State: ASSOC_DONE

Description: Waiting for outstanding call machinery to complete.

Chart: CO_CLIENT

State: ASSOC_OPEN

Description: Association available for call.

Reactions	
Trigger	**Action**
(RCV_SHUTDOWN or RESOURCES_SCARCE) [not SHUTDOWN_OK]	tr!(SHUTDOWN_REQUESTED)

Chart: CO_CLIENT

State: AUTHENTICATION

Description: Process authentication verification.

Activities Throughout:

VERIFY_AUTH

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT

State: CALL

Description: Processing a remote procedure call request.

Chart: CO_CLIENT

State: CALL_IDLE

Description: Initial remote procedure call state.

Reactions	
Trigger	**Action**
en(CALL_IDLE)	RT_CANCEL_COUNT:=0; RT_RCV_CANCEL_COUNT:=0; tr!(RT_DID_NOT_EXECUTE)

Chart: CO_CLIENT

State: CANCEL

Description: Processing of requests to terminate call in progress.

Reactions	
Trigger	**Action**
tm(CLIENT_CANCEL, TIMEOUT_CANCEL)	RT_EXCEPTION_STATUS:= CONST_RPC_S_CANCEL_TIMEOUT; st!(EXCEPTION)
tm(CLIENT_CANCEL, TIMEOUT_CANCEL)[in(STUB_WAIT)]	DEALLOC_ASSOC
tm(CLIENT_CANCEL, TIMEOUT_CANCEL)[in(REQ_WAIT)]	DEALLOC_ASSOC; ORPHANED_CALL; st!(ABORT_SEND)
tm(CLIENT_CANCEL, TIMEOUT_CANCEL)[in(CONFIRMATION)]	DEALLOC_ASSOC; ORPHANED_CALL; st!(ABORT_RECEIVE)

Chart: CO_CLIENT

State: CAN_IDLE

Description: Waits for cancel requests and handles cancel timeouts if pending cancel request.

Reactions	
Trigger	**Action**

en(CAN_IDLE)	IF
	CURRENT_PENDING_CANCEL
	THEN
	tr!(RT_PENDING_CANCEL)
	ELSE
	fs!(RT_PENDING_CANCEL)
	END IF

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT

State: CNTL_IDLE

Description: Waits for incoming control PDUs.

Reactions	
Trigger	**Action**
en(CNTL_IDLE)	fs!(FAULT_PDU); fs!(BIND_ACK_PDU); fs!(BIND_NAK_PDU); fs!(SHUTDOWN_PDU); fs!(ALTER_CONTEXT_RESP_PDU)

RECEIVE_PDU[PDU_TYPE=FAULT
and VALID_PDU_HEADER]
	tr!(FAULT_PDU);
	RT_RCV_CANCEL_COUNT:=
	PDU_CANCEL_COUNT;
	IF
	PDU_PENDING_CANCEL
	THEN
	tr!(RT_RCV_PENDING_CANCEL)
	ELSE
	fs!(RT_RCV_PENDING_CANCEL)
	END IF

RECEIVE_PDU[PDU_TYPE=BIND_ACK and VALID_PDU_HEADER]	tr!(BIND_ACK_PDU)
exiting	IF AUTH THEN RT_AUTH_VERIFIER_CNTL:= PDU_AUTH_VERIFIER; RT_AUTH_LENGTH_CNTL:= PDU_AUTH_SPEC END IF
RECEIVE_PDU[PDU_TYPE=BIND_NAK and VALID_PDU_HEADER]	tr!(BIND_NAK_PDU)
RECEIVE_PDU[PDU_TYPE=SHUTDOWN and VALID_PDU_HEADER]	tr!(SHUTDOWN_PDU)
RECEIVE_PDU[PDU_TYPE= ALTER_CONTEXT_RESP and VALID_PDU_HEADER]	tr!(ALTER_CONTEXT_RESP_PDU)
RECEIVE_PDU[CNTL_PDU and VALID_PDU_HEADER]	RCV_CNTL_PDU
RECEIVE_PDU[CNTL_PDU and not VALID_FRAG_SIZE]	RCV_FRAG_SIZE_TOO_LARGE

Chart: CO_CLIENT

State: CONFIRMATION

Description: Processing response data (**out** params) for remote procedure call.

Reactions	
Trigger	**Action**
en(CONFIRMATION)	fs!(RESPONSE_ACTIVE); RT_OUT_PARAMS:=NULL
en(CONFIRMATION)[MAYBE]	DEALLOC_ASSOC

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT

State: CONF_IDLE

Description: Receive response data from server (possibly fragmented).

Reactions	
Trigger	**Action**
RECEIVE_PDU[PDU_TYPE=RESPONSE and VALID_PDU_HEADER and not AUTH]	tr!(RESPONSE_ACTIVE); DO_RESP; RCV_FRAG_PDU
RECEIVE_PDU[PDU_TYPE=RESPONSE and VALID_PDU_HEADER and AUTH]	tr!(RESPONSE_ACTIVE); DO_RESP; RCV_FRAG_PDU_A
RECEIVE_PDU[PDU_TYPE=RESPONSE and not VALID_FRAG_SIZE]	RCV_FRAG_SIZE_TOO_LARGE
en(CONF_IDLE)[RESPONSE_ACTIVE and LAST_OUT_FRAG]	RCV_LAST_OUT_FRAG
en(CONF_IDLE)[RESPONSE_ACTIVE]	DO_OUT_PKT; st!(HANDLE_OUT_FRAG)

Chart: CO_CLIENT

State: CONN_WAIT

Description: Request transport connection and wait for response.

Reactions	
Trigger	**Action**
en(CONN_WAIT)	REQUEST_CONN

Chart: CO_CLIENT

State: CONTROL

Description: Processing received control PDUs.

Chart: CO_CLIENT

State: CO_CLIENT

Description: Main state for client association and call.

The CO_CLIENT_ALLOC machine creates this state by generating the CREATE_ASSOCIATION event.

Chart: CO_CLIENT

State: DATA

Description: Processing RPC call data.

Reactions	
Trigger	**Action**
en(DATA)	fs!(REQUEST_ACTIVE)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

State: FAULT

Description: Handling fault PDU responses.

Reactions	
Trigger	**Action**

en(FAULT)	DEALLOC_ASSOC;
	IF
	  PDU_DID_NOT_EXECUTE
	THEN
	  tr!(RT_DID_NOT_EXECUTE)
	END IF;
	RT_RCV_CANCEL_COUNT:=PDU_CANCEL_COUNT;
	RT_EXCEPTION_STATUS:=PDU_FAULT_STATUS;
	st!(EXCEPTION);
	IF
	  in(REQ_WAIT)
	THEN
	  st!(ABORT_SEND)
	END IF;
	IF
	  in(CONFIRMATION)
	THEN
	  st!(ABORT_RECEIVE)
	END IF

Chart: CO_CLIENT

State: INIT

Description: Initial state for new association. Initialise state variables.

Reactions	
Trigger	**Action**

en(INIT)	fs!(SHUTDOWN_REQUESTED);
	fs!(WAIT_FOR_GROUP);
	IF
	  GROUP_EXISTS
	THEN
	  SND_ASSOC_GROUP_ID:=RT_ASSOC_GROUP_ID
	ELSE
	  SND_ASSOC_GROUP_ID:=NULL
	END IF

Chart: CO_CLIENT

State: INIT_WAIT

Description: Wait for server's response to bind request.

Reactions	
Trigger	**Action**
RCV_BIND_ACK[not GROUP_EXISTS]	CREATE_GROUP; tr!(WAIT_FOR_GROUP)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT

State: REQ_WAIT

Description: Handles fragmented requests to server.

Reactions	
Trigger	**Action**
en(REQ_WAIT)[not REQUEST_ACTIVE]	FIRST_REQ
ex(REQ_WAIT)[not LAST_IN_FRAG]	fs!(SND_FIRST_FRAG)

Chart: CO_CLIENT

State: SND_CAN

Description: Processes cancel requests. Sends **cancel** PDU.

Reactions	
Trigger	**Action**
en(SND_CAN) or CLIENT_CANCEL	CAN_CALL; RT_CANCEL_COUNT:=RT_CANCEL_COUNT+1

Chart: CO_CLIENT

State: STUB_WAIT

Description: Wait until stub calls with first fragment.

Chart: CO_CLIENT

State: WORKING

Description: Main working state for call instance.

11.1.3 CO_CLIENT Events

The CO_CLIENT statechart defines the following events:

Chart: CO_CLIENT

Event: ABORT_ASSOC_REQ

Description: Client run-time system requested termination of association (typically local error).

Chart: CO_CLIENT

Event: ADD_TO_GROUP

Description: Signal group to add this association. Generated by an association in this group.

Association must check that the instance of the CO_CLIENT_ALLOC machine which initiated this association has not terminated. If it has terminated, then the ASSOCIATION machine must lock the group before issuing this event to avoid possible race conditions.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_GROUP.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Event: ALLOC_ASSOC_ACK

Description: Association allocated and may be used for call.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_ALLOC.

Chart: CO_CLIENT

Event: ALLOC_ASSOC_NAK

Description: Unable to allocate association because requested context not supported.

Generated in chart CO_CLIENT and sensed in CO_CLIENT_ALLOC.

Chart: CO_CLIENT

Event: ALLOC_REQ

Description: A client requested allocation of an association.

For efficiency, choose the association which has either or both a presentation context and a security context matching those requested for the call.

Event is generated by CO_CLIENT_ALLOC and sensed by CO_CLIENT.

Chart: CO_CLIENT

Event: ALTER_CONTEXT_REQ

Description: The run-time system has requested an additional presentation context negotiation.

Chart: CO_CLIENT

Event: AUTHENTICATED

Description: Authentication processing completed successfully.

Chart: CO_CLIENT

Event: CHECK_CONTEXT

Description: Check whether desired context is supported. Internally generated.

Chart: CO_CLIENT

Event: CLIENT_CANCEL

Description: The client has issued an RPC cancel request call.

Generated by the **Cancel** service primitive.

Chart: CO_CLIENT

Event: CLOSE

Description: Compound events to terminate association.

Definition:

```
ABORT_ASSOC_REQ or CONN_NAK or
 NO_CONNECTION[not in(DATA)] or
 RCV_BIND_NAK or RCV_ALTER_CONTEXT_REJECT
 or DEALLOC_ASSOC[in(ASSOC_DONE) or
 SHUTDOWN_OK and SHUTDOWN_REQUESTED] or
 (RCV_SHUTDOWN or
 RESOURCES_SCARCE)[SHUTDOWN_OK] or
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
RCV_FRAG_SIZE_TOO_LARGE[not in(DATA)] or
[in(ASSOC_DONE) and in(CALL_IDLE)] or
DENIED[in(INIT_WAIT)]
```

Chart: CO_CLIENT

Event: COMPLETE

Description: RPC call completed (with success or fault).

Definition:
```
(RCV_LAST_OUT_FRAG or DEALLOC_ASSOC or
 DENIED[not SHUTDOWN_PDU] or NO_CONNECTION
 or RCV_FRAG_SIZE_TOO_LARGE) and not
 RCV_FAULT
```

Chart: CO_CLIENT

Event: CONN_ACK

Description: Transport connection was established. Generated by underlying transport.

Chart: CO_CLIENT

Event: CONN_NAK

Description: Transport connection request failed. Generated by transport service.

Chart: CO_CLIENT

Event: CREATE_ASSOCIATION

Description: This event, generated internally, creates the CO_CLIENT machine. The presentation context and security context are passed to the association along with this event.

Event is generated by CO_CLIENT_ALLOC and sensed by CO_CLIENT.

Chart: CO_CLIENT

Event: CREATE_FAILED

Description: Failed to create a new association.

Event is generated by CO_CLIENT or CO_CLIENT_ALLOC and sensed by CO_CLIENT_ALLOC.

Chart: CO_CLIENT

Event: CREATE_GROUP

Description: Triggers creation of the association group. If this event is issued to a group that already exists, then it has no effect.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_GROUP.

Chart: CO_CLIENT

Event: CREATE_SUCCESS

Description: A new association was successfully created.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_ALLOC.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Event: DEALLOC_ASSOC

Description: Deallocation of association requested. Generated internally.

Chart: CO_CLIENT

Event: DENIED

Description: Authentication failure detected.

The VERIFY_AUTH activity generates this event if either the integrity check failed or the requested protection level for authentication services does not match.

Chart: CO_CLIENT

Event: DISCONN_REQ

Description: Request termination of transport/session connection. Generated internally.

Chart: CO_CLIENT

Event: LAST_IN_PKT

Description: Statechart internal event: last packet of fragmented request.

Definition:
```
[TRANSMIT_REQ and LAST_IN_FRAG and
 REQUEST_ACTIVE]
```

Chart: CO_CLIENT

Event: MARK_ASSOC

Description: Save status of association for RPC user and/or update presentation context set.

Because the association may terminate, the status, such as reason for termination, must be preserved for return to RPC user by the stub. This event is generated internally to CO_CLIENT.

Chart: CO_CLIENT

Event: NEXT_IN_PKT

Description: Statechart internal event: intermediate packet of fragmented request.

Definition:
```
[TRANSMIT_REQ and not LAST_IN_FRAG and
 REQUEST_ACTIVE]
```

Chart: CO_CLIENT

Event: NO_CONNECTION

Description: Notification that the underlying transport connection terminated.

Generated externally by the underlying transport service.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Event: OPEN_ASSOC

Description: Open the association for use by a call.

Definition:
```
RCV_BIND_ACK[GROUP_EXISTS] or
 [GROUP_EXISTS and WAIT_FOR_GROUP]
```

Chart: CO_CLIENT

Event: RCV_ALTER_CONTEXT_REJECT

Description: Received an **alter_context_resp** PDU marking a security integrity failure.

This failure is indicated by the data item PDU_AUTH_VALUE_SUB_TYPE set to CONST_SUB_TYPE_INVALID_CHECKSUM. This event is generated in the CNTL_CALL action (CO_CLIENT chart).

Chart: CO_CLIENT

Event: RCV_ALTER_CONTEXT_RESP

Description: Received an **alter_context_resp** PDU. Generated in CNTL_CALL action.

Chart: CO_CLIENT

Event: RCV_BIND_ACK

Description: Receive a **bind_ack** PDU. Generated in CNTL_CALL action.

Chart: CO_CLIENT

Event: RCV_BIND_NAK

Description: Received a **bind_nak** PDU. Generated in CNTL_CALL action (CO_CLIENT chart).

Chart: CO_CLIENT

Event: RCV_CNTL_PDU

Description: Received one of the control PDUs with valid header.

Chart: CO_CLIENT

Event: RCV_FAULT

Description: Received a valid **fault** PDU. Generated in CNTL_CALL action.

Chart: CO_CLIENT

Event: RCV_FRAG_PDU

Description: Received a **response** PDU for a non-authenticated call.

Chart: CO_CLIENT

Event: RCV_FRAG_PDU_A

Description: Received a **response** PDU for an authenticated call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Event: RCV_FRAG_SIZE_TOO_LARGE

Description: The received PDU exceeded the maximum allowed fragment size.

Chart: CO_CLIENT

Event: RCV_LAST_OUT_FRAG

Description: Received last fragment **response** PDU. Signal completion to stub.

The last fragment of a multi-fragmented response or a single packet response was received. RCV_LAST_OUT_FRAG signals that the complete response data is available to the stub for unmarshaling.

Chart: CO_CLIENT

Event: RCV_SHUTDOWN

Description: Shutdown indication was received from the server. Generated in CNTL_CALL action.

Chart: CO_CLIENT

Event: RECEIVE_PDU

Description: Received a PDU from server.

Chart: CO_CLIENT

Event: REMOVE_FROM_GROUP

Description: Remove association from this group.

To avoid a race condition which could result from multiple simultaneous events, the association machine must lock the group before generating the REMOVE_FROM_GROUP event and release the lock only after the event has been processed by the group machine.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_GROUP.

Chart: CO_CLIENT

Event: REQUEST_CONN

Description: Request a new transport connection. If not null, use secondary address.

Chart: CO_CLIENT

Event: RESOURCES_SCARCE

Description: Requests shutdown of association. Externally generated.

Resource management is implementation-specific. This event is generated by the implementation resource management policy when it is necessary to reclaim idle associations. It is recommended that idle associations be maintained for better performance.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.1.4 CO_CLIENT Actions

The CO_CLIENT statechart defines the following actions:

Chart: CO_CLIENT

Action: CAN_CALL

Description: Set up the **cancel** PDU to be sent.

Definition:
```
SND_REQUEST_TYPE:=CANCEL;
st!(SEND_PKT)
```

Chart: CO_CLIENT

Action: CNTL_CALL

Description: Reactions on received control PDUs. Generates respective RCV_* events.

Definition:
```
IF
   FAULT_PDU
THEN
   RCV_FAULT;
   IF
      RT_CANCEL_COUNT/=RT_RCV_CANCEL_COUNT
   THEN
      tr!(RT_PENDING_CANCEL)
   ELSE
      IF
         RT_RCV_PENDING_CANCEL
      THEN
         tr!(RT_PENDING_CANCEL)
      ELSE
         fs!(RT_PENDING_CANCEL)
      END IF
   END IF
END IF;
IF
   BIND_ACK_PDU
THEN
   RCV_BIND_ACK;
   RT_SERVER_PRES_CONTEXT_LIST:=PDU_P_RESULT_LIST;
   st!(SET_PRES_CONTEXT);
   IF
      PDU_MAX_RCV_FRAG_SIZE/=0
   THEN
      SND_MAX_XMIT_FRAG_SIZE:=PDU_MAX_RCV_FRAG_SIZE
   END IF;
   IF
      PDU_MAX_XMIT_FRAG_SIZE/=0
   THEN
      SND_MAX_RCV_FRAG_SIZE:=PDU_MAX_XMIT_FRAG_SIZE
   END IF
END IF;
IF
   ALTER_CONTEXT_RESP_PDU
THEN
   IF
      PDU_AUTH_VALUE_SUB_TYPE=
               CONST_SUB_TYPE_INVALID_CHECKSUM
   THEN
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
      RCV_ALTER_CONTEXT_REJECT
   ELSE
      RCV_ALTER_CONTEXT_RESP;
      RT_SERVER_PRES_CONTEXT_LIST:=
               RT_SERVER_PRES_CONTEXT_LIST
       +PDU_P_RESULT_LIST;
      st!(SET_PRES_CONTEXT)
   END IF
END IF;
IF
   BIND_NAK_PDU
THEN
   RCV_BIND_NAK
END IF;
IF
   SHUTDOWN_PDU
THEN
   RCV_SHUTDOWN
END IF
```

Chart: CO_CLIENT

Action: DO_OUT_PKT

Description: Append received **response** PDU body data to internal buffer.

Definition:
```
RT_OUT_PARAMS:=RT_OUT_PARAMS+RT_BODY
```

Chart: CO_CLIENT

Action: DO_RESP

Description: Evaluate **response** PDU header.

Definition:
```
RT_BODY:=PDU_BODY;
IF
   AUTH
THEN
   RT_AUTH_VERIFIER_CALL:=PDU_AUTH_VERIFIER;
   RT_AUTH_LENGTH_CALL:=PDU_AUTH_SPEC
END IF;
IF
   PDU_LAST_FRAG
THEN
   tr!(LAST_OUT_FRAG);
   IF
      PDU_CANCEL_COUNT/=RT_CANCEL_COUNT
   THEN
      tr!(RT_PENDING_CANCEL)
   ELSE
      IF
         PDU_PENDING_CANCEL
      THEN
         tr!(RT_PENDING_CANCEL)
      ELSE
         fs!(RT_PENDING_CANCEL)
      END IF
   END IF;
   RT_RCV_CANCEL_COUNT:=PDU_CANCEL_COUNT
END IF
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Action: END_ASSOC

Description: Notification that association has been closed.

Definition:

```
MARK_ASSOC;
IF
   in(INIT)
THEN
   CREATE_FAILED
END IF;
WHEN
   not NO_CONNECTION
THEN
   DISCONN_REQ
END WHEN;
IF
   in(ASSOC_OPEN) or in(ALLOCATED)
THEN
   REMOVE_FROM_GROUP
END IF
```

Chart: CO_CLIENT

Action: FINAL

Description: Perform final actions for RPC call.

Definition:

```
st!(DEQUEUE_CANCEL);
WHEN
   RCV_LAST_OUT_FRAG
THEN
   DEALLOC_ASSOC
END WHEN;
WHEN
   DENIED
THEN
   RT_EXCEPTION_STATUS:=CONST_NCA_S_INVALID_CHKSUM
END WHEN;
WHEN
   NO_CONNECTION
THEN
   RT_EXCEPTION_STATUS:=CONST_RPC_S_COMM_FAILURE;
   DEALLOC_ASSOC
END WHEN;
WHEN
   RCV_FRAG_SIZE_TOO_LARGE
THEN
   RT_EXCEPTION_STATUS:=RT_NCA_S_PROTO_ERROR
END WHEN;
WHEN
   DENIED or NO_CONNECTION or
    RCV_FRAG_SIZE_TOO_LARGE
THEN
   st!(EXCEPTION);
   IF
      not in(STUB_WAIT)
   THEN
      IF
         in(REQ_WAIT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
      THEN
         st!(ABORT_SEND)
      ELSE
         st!(ABORT_RECEIVE)
      END IF
   END IF
END WHEN
```

Chart: CO_CLIENT

Action: FIRST_REQ

Description: Set up and send first **request** PDU.

If the request is non-fragmented (single PDU), the actual send activity will be performed through the LAST_REQ action.

Definition:

```
tr!(REQUEST_ACTIVE);
tr!(SND_FIRST_FRAG);
RT_IN_FRAG:=RT_IN_PARAMS;
SND_IN_PARAMS:=RT_IN_PARAMS;
SND_PRES_CONTEXT_ID:=RT_PRES_CONTEXT_ID;
IF
   RT_OBJ_ID/=NULL
THEN
   SND_OBJ_ID:=RT_OBJ_ID
ENDIF;
SND_CALL_ID:=RT_CALL_ID;
SND_OP_NUM:=RT_OP_NUM;
IF
   not LAST_IN_FRAG
THEN
   fs!(TRANSMIT_REQ);
   fs!(RT_DID_NOT_EXECUTE);
   fs!(SND_LAST_FRAG);
   SND_REQUEST_TYPE:=REQUEST;
   st!(SEND_PKT)
END IF
```

Chart: CO_CLIENT

Action: INIT_ASSOC

Description: Initiate an association. Create **bind** PDU and send it.

Definition:

```
SND_PRES_CONTEXT_LIST:=DESIRED_CONTEXT_LIST;
SND_IF_ID:=RT_IF_ID;
SND_IF_VERSION:=RT_IF_VERSION;
SND_MAX_RCV_FRAG_SIZE:=RT_MAX_RCV_FRAG_SIZE;
SND_MAX_XMIT_FRAG_SIZE:=RT_MAX_XMIT_FRAG_SIZE;
SND_REQUEST_TYPE:=BIND;
st!(SEND_PKT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Action: LAST_REQ

Description: Send last **in** fragment to server.

Definition:
```
fs!(TRANSMIT_REQ);
fs!(RT_DID_NOT_EXECUTE);
tr!(SND_LAST_FRAG);
SND_IN_PARAMS:=RT_IN_FRAG;
SND_REQUEST_TYPE:=REQUEST;
st!(SEND_PKT)
```

Chart: CO_CLIENT

Action: MAKE_AVAIL

Description: Make the association available for a call.

Definition:
```
CREATE_SUCCESS;
ADD_TO_GROUP
```

Chart: CO_CLIENT

Action: NEXT_REQ

Description: Send next **in** fragment to server.

Definition:
```
fs!(TRANSMIT_REQ);
SND_IN_PARAMS:=RT_IN_FRAG;
SND_REQUEST_TYPE:=REQUEST;
st!(SEND_PKT)
```

Chart: CO_CLIENT

Action: ORPHANED_CALL

Description: Set up **orphaned** PDU to be sent.

Definition:
```
SND_REQUEST_TYPE:=ORPHANED;
st!(SEND_PKT)
```

11.1.5 CO_CLIENT Conditions

The CO_CLIENT statechart defines the following conditions:

Chart: CO_CLIENT

Condition: ALTER_CONTEXT_RESP_PDU

Description: Statechart internal flag: received PDU type **alter_context_resp**.

Chart: CO_CLIENT

Condition: AUTH

Description: Statechart internal flag: indicates that call is authenticated.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Condition: BIND_ACK_PDU

Description: Statechart internal flag: received PDU type **bind_ack**.

Chart: CO_CLIENT

Condition: BIND_NAK_PDU

Description: Statechart internal flag: received PDU type **bind_nak**.

Chart: CO_CLIENT

Condition: CHECK_AUTH

Description: Verify authentication if requested (not required for **shutdown** and **bind_nak** PDUs).

Definition:
```
AUTH and (not PDU_TYPE=SHUTDOWN and not
 PDU_TYPE=BIND_NAK)
```

Chart: CO_CLIENT

Condition: CNTL_PDU

Description: Statechart internal flag: to be received control PDUs.

Definition:
```
PDU_TYPE=FAULT or PDU_TYPE=BIND_ACK or
 PDU_TYPE=BIND_NAK or PDU_TYPE=SHUTDOWN or
 PDU_TYPE=ALTER_CONTEXT_RESP
```

Chart: CO_CLIENT

Condition: CURRENT_PENDING_CANCEL

Description: Cancel pending state passed from stub during initialisation of call.

Chart: CO_CLIENT

Condition: FAULT_PDU

Description: Statechart internal flag: received PDU type **fault**.

Chart: CO_CLIENT

Condition: GROUP_EXISTS

Description: The group to which this association belongs exists.

Definition:
```
in(CO_CLIENT_GROUP:CO_CLIENT_GROUP)
```

Chart: CO_CLIENT

Condition: LAST_IN_FRAG

Description: Statechart internal flag: last **in** fragment or non-frag **in** packet ready to send.

This flag is set if the transmit queue contains the last fragment (see also Subclause 9.3 on page 304).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT
Condition: LAST_OUT_FRAG
Description: Statechart internal flag: last **out** fragment or non-frag **out** packet received.

Chart: CO_CLIENT
Condition: MAYBE
Description: Statechart internal flag: maybe call.

Chart: CO_CLIENT
Condition: PDU_DID_NOT_EXECUTE
Description: **fault** PDU header flag PFC_DID_NOT_EXECUTE.

Chart: CO_CLIENT
Condition: PDU_LAST_FRAG
Description: Header flag PFC_LAST_FRAG.

Chart: CO_CLIENT
Condition: PDU_PENDING_CANCEL
Description: Header flag PFC_PENDING_CANCEL in received **response** or **fault** PDU.

Chart: CO_CLIENT
Condition: PRES_SEC_CONTEXT_SUPPORTED
Description: The presentation and security contexts for the call are in the negotiated set.

Both the negotiated presentation context and the security credentials are saved by the SET_PRES_SEC_CONTEXT activity.

Chart: CO_CLIENT
Condition: REQUEST_ACTIVE
Description: Statechart internal flag: send request has started.

Chart: CO_CLIENT
Condition: RESPONSE_ACTIVE
Description: Statechart internal flag: indicates availability of response data.

Chart: CO_CLIENT
Condition: RT_DID_NOT_EXECUTE
Description: Run time internal: the call has not been executed yet by the server manager.

This flag is common state shared between the stub and the run-time system. The stub must initialise this to true. The run-time system updates this flag. If a call fails, the stub may check this flag to ascertain whether it may safely retry the call when exactly-once semantics were requested.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT
Condition: RT_PENDING_CANCEL
Description: Statechart internal flag: cancel pending state at server.

Chart: CO_CLIENT
Condition: RT_RCV_PENDING_CANCEL
Description: Statechart internal: holds received pending cancel state.

Chart: CO_CLIENT
Condition: SHUTDOWN_OK
Description: Shutdown of association allowed.

Association must lock group when checking these state variables.

Definition: `ASSOC_COUNT>1 or ACTIVE_CONTEXT_COUNT=0`

Chart: CO_CLIENT
Condition: SHUTDOWN_PDU
Description: Statechart internal flag: received PDU type **shutdown**.

Chart: CO_CLIENT
Condition: SHUTDOWN_REQUESTED
Description: Orderly shutdown of association requested.

Chart: CO_CLIENT
Condition: SND_FIRST_FRAG
Description: Statechart internal flag: header flag (PFC_FIRST_FRAG) of next frag to be sent.

Chart: CO_CLIENT
Condition: SND_LAST_FRAG
Description: Statechart internal flag: header flag (PFC_LAST_FRAG) of next buffered fragment.

Chart: CO_CLIENT
Condition: TRANSMIT_REQ
Description: One or more fragments are queued for transmission of request data.

This flag indicates that one or more request fragment(s) are queued in a run-time internal buffer and ready to be transmitted.

The run-time system internally sets this flag (true) after the stub initially provides data in the transmit queue, sufficient for at least the first PDU fragment to be transmitted. The protocol machine resets this flag if it has detected and taken an event for sending the next fragment in the queue. The run-time system sets this flag again after completion of a SEND_PKT activity if the transmit queue contains enough data for the next PDU fragment to be transmitted.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT

Condition: VALID_FRAG_SIZE

Description: Evaluation whether received PDU exceeds size limit.

Definition: `PDU_FRAG_LENGTH<=RT_MAX_RCV_FRAG_SIZE`

Chart: CO_CLIENT

Condition: VALID_PDU_HEADER

Description: Pre-evaluation of PDU header (before authentication processing).

Definition:
```
PDU_CALL_ID=RT_CALL_ID and
 PDU_VERSION_NUM=CO_VERSION_NUM_V20 and
 PDU_VERSION_NUM_MINOR=CO_VERSION_NUM_V20_MINOR
 and VALID_FRAG_SIZE
```

Chart: CO_CLIENT

Condition: WAIT_FOR_GROUP

Description: Indicates association is waiting for group to be created.

11.1.6 CO_CLIENT Data-Items

The CO_CLIENT statechart defines the following data items:

Chart: CO_CLIENT

Data Item: ACTIVE_CONTEXT_COUNT

Description: Number of active context handles for group. State variable of CO_CLIENT_GROUP.

Chart: CO_CLIENT

Data Item: ALTER_CONTEXT

Description: Constant: PDU type **alter_context**.

Definition: 14

Chart: CO_CLIENT

Data Item: ALTER_CONTEXT_RESP

Description: Constant: PDU type **alter_context_resp**.

Definition: 15

Chart: CO_CLIENT

Data Item: ASSOC_COUNT

Description: Number of associations in group. State variable of CO_CLIENT_GROUP.
Must lock group before accessing this state variable to avoid race conditions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT
Data Item: BIND
Description: Constant: PDU type **bind**.
Definition: 11

Chart: CO_CLIENT
Data Item: BIND_ACK
Description: Constant: PDU type **bind_ack**.
Definition: 12

Chart: CO_CLIENT
Data Item: BIND_NAK
Description: Constant: PDU type **bind_nak**.
Definition: 13

Chart: CO_CLIENT
Data Item: CANCEL
Description: Constant: PDU type **cancel**.
Definition: 18

Chart: CO_CLIENT
Data Item: CONST_NCA_S_INVALID_CHKSUM
Description: Constant: fault status code nca_s_invalid_chksum.

Chart: CO_CLIENT
Data Item: CONST_RPC_S_CANCEL_TIMEOUT
Description: Constant: fault status code rpc_s_cancel_timeout.

Chart: CO_CLIENT
Data Item: CONST_RPC_S_COMM_FAILURE
Description: Constant: fault status code rpc_s_comm_failure.

Chart: CO_CLIENT
Data Item: CONST_SUB_TYPE_INVALID_CHECKSUM
Description: Value indicating a security integrity failure (invalid checksum).

The value dce_c_cn_dce_sub_type_invalid_checksum, which is encoded in the **sub_type** field of the **auth_value** member of the authentication verifier. (See Section 13.)

Definition: 2

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT
Data Item: CO_VERSION_NUM_V20
Description: Constant: RPC protocol version 2.0 major version number.
Definition: 5

Chart: CO_CLIENT
Data Item: CO_VERSION_NUM_V20_MINOR
Description: Constant: RPC protocol minor version number.

Chart: CO_CLIENT
Data Item: DESIRED_CONTEXT_LIST
Description: Presentation context determined by stub.

This is the presentation context required for the call. Its value is determined by the stub from the interface definition and transfer syntaxes. See the subclause on PDU encodings for details of this context.

Chart: CO_CLIENT
Data Item: FAULT
Description: Constant: PDU type **fault**.
Definition: 3

Chart: CO_CLIENT
Data Item: ORPHANED
Description: Constant: PDU type **orphaned**.
Definition: 19

Chart: CO_CLIENT
Data Item: PDU_AUTH_SPEC
Description: PDU header field **auth_length**.

Chart: CO_CLIENT
Data Item: PDU_AUTH_VALUE_SUB_TYPE
Description: The value of the **sub_type** field of the **auth_value** member of the authentication verifier as received in an **alter_context_resp** PDU. (See Section 13.)

Chart: CO_CLIENT
Data Item: PDU_AUTH_VERIFIER
Description: PDU trailer: authentication verifier (authentication protocol-specific).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT
Data Item: PDU_BODY
Description: Array of PDU body data.

Chart: CO_CLIENT
Data Item: PDU_CALL_ID
Description: PDU header field **call_id**.

Chart: CO_CLIENT
Data Item: PDU_CANCEL_COUNT
Description: Received cancel_count value in **response** or **fault** PDU header.

Chart: CO_CLIENT
Data Item: PDU_FAULT_STATUS
Description: Constant: fault status code.

Chart: CO_CLIENT
Data Item: PDU_FRAG_LENGTH
Description: PDU header field **frag_length**.

Chart: CO_CLIENT
Data Item: PDU_MAX_RCV_FRAG_SIZE
Description: PDU header field **max_recv_frag**.

Chart: CO_CLIENT
Data Item: PDU_MAX_XMIT_FRAG_SIZE
Description: PDU header field **max_xmit_frag**.

Chart: CO_CLIENT
Data Item: PDU_P_RESULT_LIST
Description: PDU header field: **p_result_list** in **bind_ack** and **alter_context_resp** PDUs.

Chart: CO_CLIENT
Data Item: PDU_TYPE
Description: PDU header field **PTYPE**.

Chart: CO_CLIENT
Data Item: PDU_VERSION_NUM
Description: PDU header field **rpc_vers**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT
Data Item: PDU_VERSION_NUM_MINOR
Description: PDU header field **rpc_vers_minor**.

Chart: CO_CLIENT
Data Item: REQUEST
Description: Constant: PDU type **request**.
Definition: 0

Chart: CO_CLIENT
Data Item: RESPONSE
Description: Constant: PDU type **response**.
Definition: 2

Chart: CO_CLIENT
Data Item: RT_ASSOC_GROUP_ID
Description: Group ID for newly created association. Defined in CO_CLIENT_GROUP.

Chart: CO_CLIENT
Data Item: RT_AUTH_LENGTH_CALL
Description: Statechart internal: authentication length field received in CALL state.

Chart: CO_CLIENT
Data Item: RT_AUTH_LENGTH_CNTL
Description: Statechart internal: authentication length field received in CONTROL state.

Chart: CO_CLIENT
Data Item: RT_AUTH_VERIFIER_CALL
Description: Received authentication trailer (verifier) for **request** PDU.

Chart: CO_CLIENT
Data Item: RT_AUTH_VERIFIER_CNTL
Description: Received authentication trailer (verifier) for **control** PDU.

Chart: CO_CLIENT
Data Item: RT_BODY
Description: Statechart internal: temporarily buffered **response** PDU body data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Data Item: RT_CALL_ID

Description: Statechart internal: call identifier of current RPC call.

Chart: CO_CLIENT

Data Item: RT_CANCEL_COUNT

Description: Statechart internal counter: number of cancel requests sent.

Chart: CO_CLIENT

Data Item: RT_EXCEPTION_STATUS

Description: Statechart internal: status value passed to exception handler.

Chart: CO_CLIENT

Data Item: RT_IF_ID

Description: Statechart internal: received interface UUID of call.

Chart: CO_CLIENT

Data Item: RT_IF_VERSION

Description: Statechart internal: received interface version number.

Chart: CO_CLIENT

Data Item: RT_IN_FRAG

Description: Statechart internal pointer to data to be sent in next **request** PDU.

The SEND_PKT activity increments this pointer after a **request** PDU is sent.

Chart: CO_CLIENT

Data Item: RT_IN_PARAMS

Description: Statechart internal: buffered array of reassembled input data.

RT_IN_PARAMS is the queue of transmit data provided by the stub. A possible segmentation of this queue is not equivalent to the sizes of PDU fragments sent by the run-time system (SEND_PKT) activity.

The RT_IN_FRAG variable is a pointer data type that points to the to be transmitted data fragment within this RT_IN_PARAMS queue.

Chart: CO_CLIENT

Data Item: RT_MAX_RCV_FRAG_SIZE

Description: Maximum size of a fragment the receiver is able to handle.

The minimum value of this fragment size is determined by the architected value MustRcvFragSize (refer to Subclause 12.6.2 on page 476).

Implementations may support larger fragment sizes that are subject to negotiation with the server. This value is set internally by run-time implementations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Data Item: RT_MAX_XMIT_FRAG_SIZE

Description: Maximum size of a fragment the sender is able to handle.

The minimum value of this fragment size is determined by the architected value MustRcvFragSize (refer to Subclause 12.6.2 on page 476).

Implementations may support larger fragment sizes that are subject to negotiation with the server. This value is set internally by run-time implementations.

Chart: CO_CLIENT

Data Item: RT_NCA_S_PROTO_ERROR

Description: Constant: fault status code nca_s_proto_error.

Chart: CO_CLIENT

Data Item: RT_OBJ_ID

Description: Statechart internal: buffered object UUID of RPC call.

Chart: CO_CLIENT

Data Item: RT_OP_NUM

Description: Statechart internal: buffered operation number of RPC call.

Chart: CO_CLIENT

Data Item: RT_OUT_PARAMS

Description: Buffered array of unfragmented output data.

Chart: CO_CLIENT

Data Item: RT_PRES_CONTEXT_ID

Description: Statechart internal: presentation context identifier of current call.

Selection of values for the context identifier is implementation-dependent. There must be a one-to-one mapping between each negotiated context and the identifiers within an association.

Chart: CO_CLIENT

Data Item: RT_RCV_CANCEL_COUNT

Description: Statechart internal: received cancel count.

Chart: CO_CLIENT

Data Item: RT_SERVER_PRES_CONTEXT_LIST

Description: Statechart internal: the received set of supported server presentation contexts.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT
Data Item: SHUTDOWN
Description: Constant: PDU type **shutdown**.
Definition: 17

Chart: CO_CLIENT
Data Item: SND_ASSOC_GROUP_ID
Description: Association group ID sent in **bind** and **alter_context** PDUs.

Chart: CO_CLIENT
Data Item: SND_CALL_ID
Description: Call ID of current RPC call.

Chart: CO_CLIENT
Data Item: SND_IF_ID
Description: Interface UUID to be sent.

Chart: CO_CLIENT
Data Item: SND_IF_VERSION
Description: Interface version number to be sent.

Chart: CO_CLIENT
Data Item: SND_IN_PARAMS
Description: PDU body data promoted to SEND_PKT activity.

Chart: CO_CLIENT
Data Item: SND_MAX_RCV_FRAG_SIZE
Description: Constant: Maximum receive fragment size. Sent in **bind** PDU.

Chart: CO_CLIENT
Data Item: SND_MAX_XMIT_FRAG_SIZE
Description: Constant: Maximum transmit fragment size. Sent in **bind** PDU.

Chart: CO_CLIENT
Data Item: SND_OBJ_ID
Description: Object UUID of current RPC call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT

Data Item: SND_OP_NUM

Description: Operation number of current call.

Chart: CO_CLIENT

Data Item: SND_PRES_CONTEXT_ID

Description: Determined by the presentation context in the binding information.

Chart: CO_CLIENT

Data Item: SND_PRES_CONTEXT_LIST

Description: Presentation context list to be sent.

Chart: CO_CLIENT

Data Item: SND_REQUEST_TYPE

Description: PDU type to be sent.

Chart: CO_CLIENT

Data Item: TIMEOUT_CANCEL

Description: Timeout value for cancel requests.

Sets the lower bound on the time to wait before timing out after forwarding a cancel PDU to the server. The default of this timeout value is set to infinity (see Annex K). Applications may set a different value via the **rpc_mgmt_set_cancel_timeout** RPC API.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.2 CO_CLIENT_ALLOC Machine

Figure 11-2 shows the CO_CLIENT_ALLOC machine statechart.

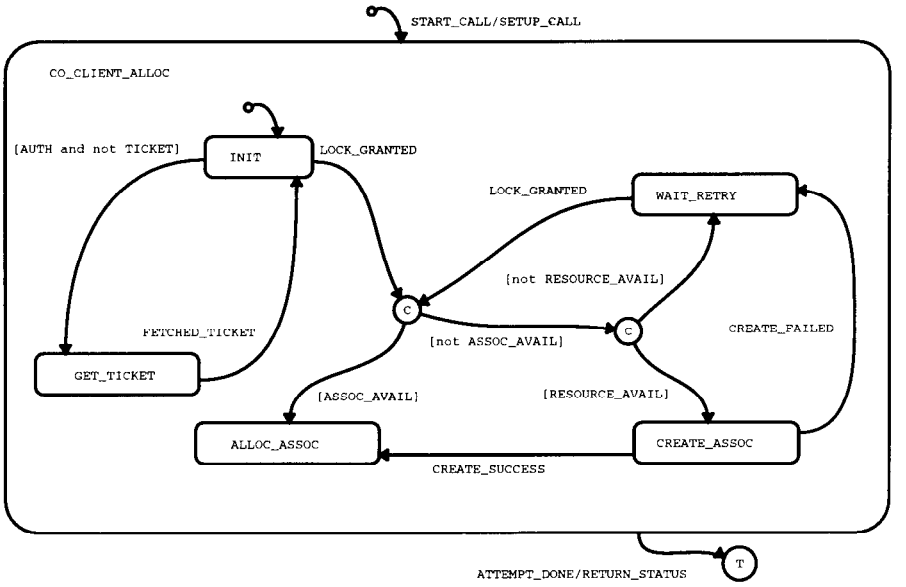


Figure 11-2 CO_CLIENT_ALLOC Statechart

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.2.1 CO_CLIENT_ALLOC Activities

The CO_CLIENT_ALLOC statechart defines the following activities:

Chart: CO_CLIENT_ALLOC

Activity: FETCH_TICKET

Description: Obtains the security context for the RPC session from the security service (that is, kerberos ticket, if authentication service is rpc_c_authn_dce_secret).

The activity resets the conditional flag TICKET to false at the beginning, and sets TICKET to true before termination only if the fetch operation succeeded. FETCH_TICKET is a self-terminating activity.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.2.2 CO_CLIENT_ALLOC States

The CO_CLIENT_ALLOC statechart defines the following states:

Chart: CO_CLIENT_ALLOC

State: ALLOC_ASSOC

Description: Wait for association to be allocated.

Reactions	
Trigger	**Action**
en(ALLOC_ASSOC)	ALLOC_REQ

Chart: CO_CLIENT_ALLOC

State: CO_CLIENT_ALLOC

Description: Protocol machine for association allocation. Created by **Invoke** service primitive.

Chart: CO_CLIENT_ALLOC

State: CREATE_ASSOC

Description: Create a new association.

Reactions	
Trigger	**Action**

en(CREATE_ASSOC)	IF
	in(CO_CLIENT_GROUP:CO_CLIENT_GROUP)
	and RT_SECONDARY_ADDRESS=NULL
	THEN
	CREATE_FAILED
	ELSE
	CREATE_ASSOCIATION
	END IF

Chart: CO_CLIENT_ALLOC

State: GET_TICKET

Description: Get authentication ticket from security server (security service-specific).

Activities Throughout:
FETCH_TICKET

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_CLIENT_ALLOC

State: INIT

Description: Initialise state for a call.

Reactions	
Trigger	**Action**
en(INIT)[not AUTH or TICKET]	TIMEOUT_MAX_ALLOC_WAIT:=3; MAP_TO_GROUP_AND_LOCK; TIMEOUT_RANDOM:=RAND_UNIFORM(0,3)

Chart: CO_CLIENT_ALLOC

State: WAIT_RETRY

Description: Wait and retry if resources for association currently not available.

Reactions	
Trigger	**Action**
en(WAIT_RETRY)	UNLOCK_GROUP; TIMEOUT_RANDOM:= MIN(RAND_UNIFORM (0,TIMEOUT_MAX_ALLOC_WAIT), CONST_MAX_BACKOFF); TIMEOUT_MAX_ALLOC_WAIT:= 2*TIMEOUT_MAX_ALLOC_WAIT
tm(en(WAIT_RETRY), TIMEOUT_RANDOM)	MAP_TO_GROUP_AND_LOCK

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.2.3 CO_CLIENT_ALLOC Events

The CO_CLIENT_ALLOC statechart defines the following events:

Chart: CO_CLIENT_ALLOC

Event: ALLOC_ASSOC_ACK

Description: Association allocated and may be used for call.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_ALLOC.

Chart: CO_CLIENT_ALLOC

Event: ALLOC_ASSOC_NAK

Description: Unable to allocate association because requested context not supported.

Generated in chart CO_CLIENT and sensed in CO_CLIENT_ALLOC.

Chart: CO_CLIENT_ALLOC

Event: ALLOC_REQ

Description: A client requested allocation of an association.

For efficiency, choose the association which has either or both a presentation context and a security context matching those requested for the call.

Event is generated by CO_CLIENT_ALLOC and sensed by CO_CLIENT.

Chart: CO_CLIENT_ALLOC

Event: ATTEMPT_DONE

Description: Attempt to allocate association either completed successfully or failed.

Definition:

```
RCV_BIND_NAK or RCV_ALTER_CONTEXT_REJECT
 or ALLOC_ASSOC_ACK or ALLOC_ASSOC_NAK or
 tm(en(INIT),
 CONST_MAX_RESOURCE_WAIT)[not
 in(ALLOC_ASSOC)] or NO_COMMUNICATION or
 sp(FETCH_TICKET)[not TICKET]
```

Chart: CO_CLIENT_ALLOC

Event: CREATE_ASSOCIATION

Description: This event, generated internally, creates the CO_CLIENT machine.

The presentation context and security context are passed to the association along with this event.

Event is generated by CO_CLIENT_ALLOC and sensed by CO_CLIENT.

Chart: CO_CLIENT_ALLOC

Event: CREATE_FAILED

Description: Failed to create a new association.

Event is generated by CO_CLIENT or CO_CLIENT_ALLOC and sensed by CO_CLIENT_ALLOC.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT_ALLOC

Event: CREATE_SUCCESS

Description: A new association was successfully created.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_ALLOC.

Chart: CO_CLIENT_ALLOC

Event: FETCHED_TICKET

Description: Client fetched a valid authentication ticket.

Definition: `sp(FETCH_TICKET)[TICKET]`

Chart: CO_CLIENT_ALLOC

Event: LOCK_GRANTED

Description: The request to lock access to the group was granted.

To guard against race conditions, the model assumes the existence of a centralised locking mechanism for each group. One lock is associated with each group. Requests for a lock are queued and serviced in FIFO order. At most one machine may be in possession of the lock for a particular group at any time.

Chart: CO_CLIENT_ALLOC

Event: MAP_TO_GROUP_AND_LOCK

Description: Map the call to a group based upon the binding information and request lock.

Determine the group to which the binding information maps. Request a lock for this group. If the group does not exist, then possession of the lock indicates that this allocation machine is permitted to create the group.

Chart: CO_CLIENT_ALLOC

Event: NO_COMMUNICATION

Description: An unrecoverable network error occurred. Generated by underlying transport.

This may occur either during an attempt to create a new association or during an attempt to allocate an existing association.

Chart: CO_CLIENT_ALLOC

Event: RCV_ALTER_CONTEXT_REJECT

Description: Received an **alter_context_resp** PDU marking a security integrity failure.

This failure is indicated by data item PDU_AUTH_VALUE_SUB_TYPE set to CONST_SUB_TYPE_INVALID_CHECKSUM. This event is generated in the CNTL_CALL action (CO_CLIENT chart).

Chart: CO_CLIENT_ALLOC

Event: RCV_BIND_NAK

Description: Received a **bind_nak** PDU. Generated in CNTL_CALL action (CO_CLIENT chart).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_CLIENT_ALLOC

Event: START_CALL

Description: The client has initiated a remote procedure call (**Invoke** service primitive).

Chart: CO_CLIENT_ALLOC

Event: UNLOCK_GROUP

Description: Release group lock or dequeue pending lock request. Generated internally.

If the machine has been granted the lock for the group, then release the lock. If the machine has queued a request for the lock, then dequeue the request.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.2.4 CO_CLIENT_ALLOC Actions

The CO_CLIENT_ALLOC statechart defines the following actions:

Chart: CO_CLIENT_ALLOC

Action: RETURN_STATUS

Description: Return status of allocation attempt to stub and unlock group.

The CO_CLIENT_ALLOC machine indicates to the stub whether an association was allocated. If an association was successfully allocated, then a the identity of the association is returned to the stub. If the allocation attempt failed, then the reason for the failure is returned to the stub.

Definition: `UNLOCK_GROUP`

Chart: CO_CLIENT_ALLOC

Action: SETUP_CALL

Description: Set up and initialise call data.

Definition: `tr!(RT_DID_NOT_EXECUTE)`

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.2.5 CO_CLIENT_ALLOC Conditions

The CO_CLIENT_ALLOC statechart defines the following conditions:

Chart: CO_CLIENT_ALLOC

Condition: ASSOC_AVAIL

Description: The group exists and at least one association in the group is available.

The group indicated by the binding information exists, and at least one association in that group is in the ASSOC_OPEN state.

Definition: `in(CO_CLIENT:ASSOC_OPEN)`

Chart: CO_CLIENT_ALLOC

Condition: AUTH

Description: Statechart internal flag: indicates that call is authenticated.

Chart: CO_CLIENT_ALLOC

Condition: RESOURCE_AVAIL

Description: Policy and resources permit new association. Value determined externally.

The policy for allowing creation of new associations and the management of resources is implementation-dependent.

Chart: CO_CLIENT_ALLOC

Condition: RT_DID_NOT_EXECUTE

Description: Run time internal: the call has not been executed yet by the server manager.

This flag is common state shared between the stub and the run-time system. The stub must initialise this to true. The run-time system updates this flag. If a call fails, the stub may check this flag to ascertain whether it may safely retry the call when exactly-once semantics were requested.

Chart: CO_CLIENT_ALLOC

Condition: TICKET

Description: The authentication ticket is valid. Determined externally.

The authentication ticket from the call's client principal to the server's principal is valid. The particular ticket depends on the client/server pair of principals, and may be different for different RPCs.

Note that implementations may cache unexpired tickets, even across process invocations or system reboots. Therefore, this condition predicate may be maintained externally to the RPC run-time system.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.2.6 CO_CLIENT_ALLOC Data-Items

The CO_CLIENT_ALLOC statechart defines the following data items:

Chart: CO_CLIENT_ALLOC
Data Item: CONST_MAX_BACKOFF
Description: Upper bound in seconds for wait to retry allocation. Architectural constant.
Definition: 60

Chart: CO_CLIENT_ALLOC
Data Item: CONST_MAX_RESOURCE_WAIT
Description: Maximum time in seconds to wait for association allocation. Architected value.
Definition: 300

Chart: CO_CLIENT_ALLOC
Data Item: RT_SECONDARY_ADDRESS
Description: The secondary address for the group. Determined by CO_CLIENT_GROUP.

Chart: CO_CLIENT_ALLOC
Data Item: TIMEOUT_MAX_ALLOC_WAIT
Description: Maximum wait before retrying association allocation. Internal variable.

Chart: CO_CLIENT_ALLOC
Data Item: TIMEOUT_RANDOM
Description: Random time between 0 and TIMEOUT_MAX_ALLOC_WAIT. Internal variable.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.3 CO_CLIENT_GROUP Machine

Figure 11-3 shows the CO_CLIENT_GROUP machine statechart.

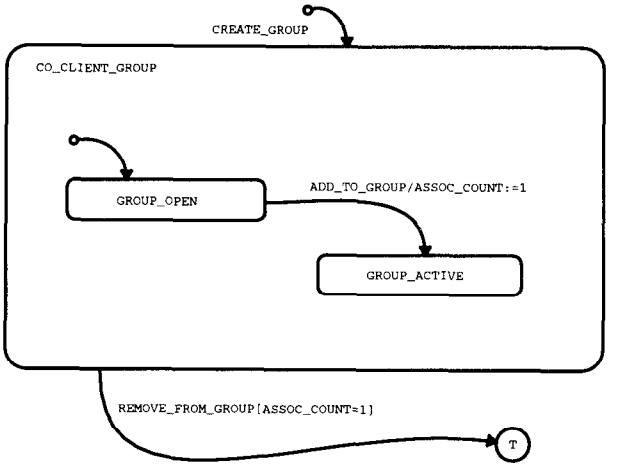


Figure 11-3 CO_CLIENT_GROUP Statechart

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.3.1 CO_CLIENT_GROUP States

The CO_CLIENT_GROUP statechart defines the following states:

Chart: CO_CLIENT_GROUP

State: CO_CLIENT_GROUP

Description: Client association group machine. Created by CO_CLIENT:CREATE_GROUP event.

The group is created when the CO_CLIENT receives a valid **bind_ack** PDU and the group did not already exist. Note that once the CO_CLIENT_GROUP is terminated, the group ID, RT_ASSOC_GROUP_ID, is no longer valid. If the run-time system or stub was maintaining this value with other state (for example, it may be stored with other binding data pointed to by a binding handle) then the value must be invalidated appropriately.

Chart: CO_CLIENT_GROUP

State: GROUP_ACTIVE

Description: Associations belong to this group.

Reactions	
Trigger	**Action**
REMOVE_FROM_GROUP [ASSOC_COUNT>1]	ASSOC_COUNT:=ASSOC_COUNT-1
ADD_TO_GROUP	ASSOC_COUNT:=ASSOC_COUNT+1
CONTEXT_ACTIVE	ACTIVE_CONTEXT_COUNT:= ACTIVE_CONTEXT_COUNT+1
CONTEXT_INACTIVE	ACTIVE_CONTEXT_COUNT:= ACTIVE_CONTEXT_COUNT-1

Chart: CO_CLIENT_GROUP

State: GROUP_OPEN

Description: Open a new associations group.

Reactions	
Trigger	**Action**
en(GROUP_OPEN)	ACTIVE_CONTEXT_COUNT:=0; ASSOC_COUNT:=0; RT_ASSOC_GROUP_ID:=PDU_ASSOC_GROUP_ID; RT_SECONDARY_ADDRESS:=PDU_SEC_ADDR

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.3.2 CO_CLIENT_GROUP Events

The CO_CLIENT_GROUP statechart defines the following events:

Chart: CO_CLIENT_GROUP

Event: ADD_TO_GROUP

Description: Signal group to add this association. Generated by an association in this group.

Association must check that the instance of the CO_CLIENT_ALLOC machine which initiated this association has not terminated. If it has terminated, then the ASSOCIATION machine must lock the group before issuing this event to avoid possible race conditions.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_GROUP.

Chart: CO_CLIENT_GROUP

Event: CONTEXT_ACTIVE

Description: A context handle was activated. Generated by the client stub.

The stub generates this event for each context handle which makes a transition from inactive to active. To avoid a race condition which could result from multiple simultaneous events, the stub must lock the group before generating the CONTEXT_ACTIVE event and release the lock only after the event has been processed by the group machine.

Chart: CO_CLIENT_GROUP

Event: CONTEXT_INACTIVE

Description: Context handle deactivated. Generated by the client stub.

The stub generates this event for each context handle which makes a transition from active to inactive. To avoid a race condition which could result from multiple simultaneous events, the stub must lock the group before generating the CONTEXT_INACTIVE event and release the lock only after the event has been processed by the group machine.

Chart: CO_CLIENT_GROUP

Event: CREATE_GROUP

Description: Triggers creation of the association group. If this event is issued to a group that already exists, then it has no effect.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_GROUP.

Chart: CO_CLIENT_GROUP

Event: REMOVE_FROM_GROUP

Description: Remove association from this group.

To avoid a race condition which could result from multiple simultaneous events, the association machine must lock the group before generating the REMOVE_FROM_GROUP event and release the lock only after the event has been processed by the group machine.

Event is generated by CO_CLIENT and sensed by CO_CLIENT_GROUP.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.3.3 CO_CLIENT_GROUP Data-Items

The CO_CLIENT_GROUP statechart defines the following data items:

Chart: CO_CLIENT_GROUP
Data Item: ACTIVE_CONTEXT_COUNT
Description: Number of active context handles for group. State variable of CO_CLIENT_GROUP.

Chart: CO_CLIENT_GROUP
Data Item: ASSOC_COUNT
Description: Number of associations in group. State variable of CO_CLIENT_GROUP.
Must lock group before accessing this state variable to avoid race conditions.

Chart: CO_CLIENT_GROUP
Data Item: PDU_ASSOC_GROUP_ID
Description: The association group ID (header field **assoc_group_id**) from the **bind_ack** PDU.

Chart: CO_CLIENT_GROUP
Data Item: PDU_SEC_ADDR
Description: The optional secondary address (header field **sec_addr**) from the **bind_ack** PDU.

Chart: CO_CLIENT_GROUP
Data Item: RT_ASSOC_GROUP_ID
Description: Group ID for newly created association. Defined in CO_CLIENT_GROUP.

Chart: CO_CLIENT_GROUP
Data Item: RT_SECONDARY_ADDRESS
Description: The secondary address for the group. Determined by CO_CLIENT_GROUP.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.4 CO_SERVER Machine

Figure 11-4 shows the CO_SERVER machine statechart.

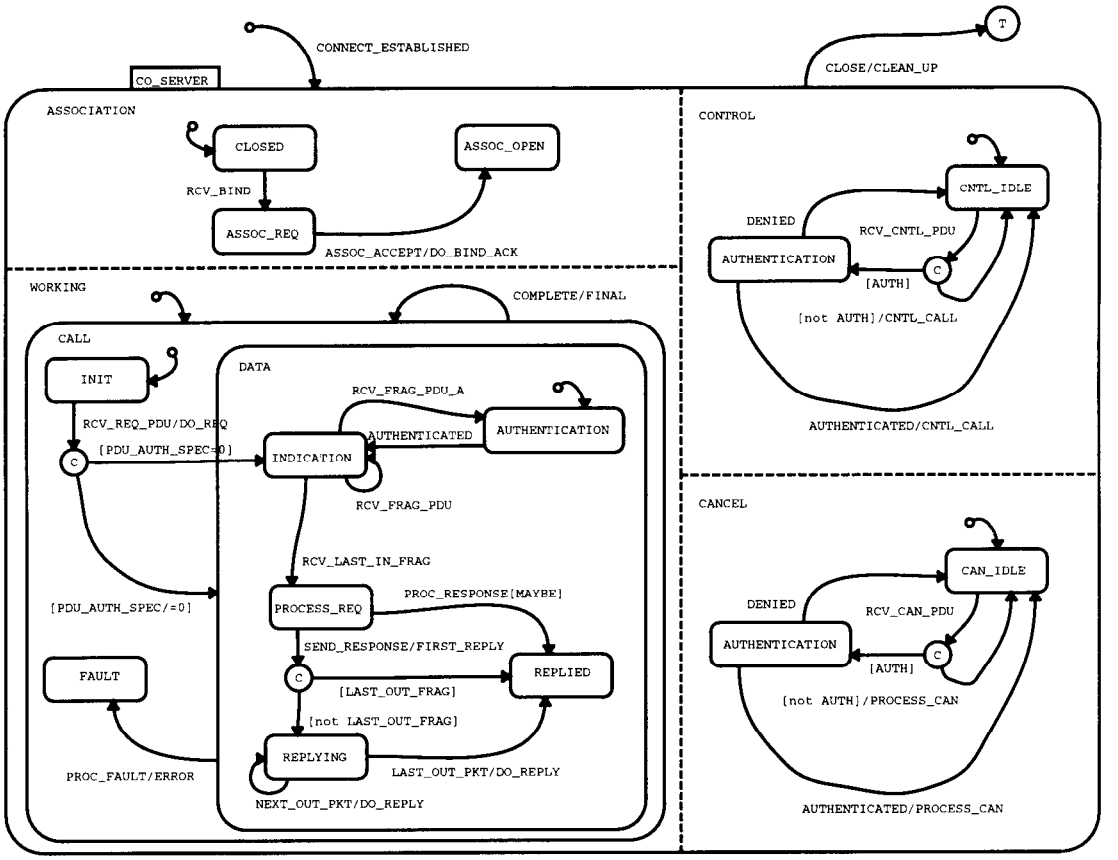


Figure 11-4 CO_SERVER Statechart

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.4.1 CO_SERVER Activities

The CO_SERVER statechart defines the following activities:

Chart: CO_SERVER

Activity: ABORT_RECEIVE

Description: Flush and discard any further received packets for this call. There may be numerous additional packets in the pipeline. The flush may be lazy, upon subsequent receive processing. Also, notify the run-time system and stub to reclaim any resources for this call.

Chart: CO_SERVER

Activity: ABORT_SEND

Description: Discontinue any further transmission of response data for the current call, to the best extent possible. Some error condition has caused a fault.

Chart: CO_SERVER

Activity: CANCEL_NOTIFY_APP

Description: This activity notifies the manager routine of the RPC application about the cancel request issued by the client.

CANCEL_NOTIFY_APP activity terminates after acknowledgement from the stub. The stub sets the RETURN_PENDING_CANCEL flag appropriately.

Chart: CO_SERVER

Activity: HANDLE_IN_FRAG

Description: This activity is invoked at each received fragment evaluation of **in** parameters for multi-fragmented RPC requests.

The HANDLE_IN_FRAG activity makes received data of the next fragment available to the stub for unmarshalling and passes the object UUID (RT_OBJ_ID) to the manager routine. This does not require a transfer of control from the run-time system to the stub for each fragment; implementation policy determines when control is transferred.

Chart: CO_SERVER

Activity: SEND_PKT

Description: Prepare a PDU to send to the client, adding the appropriate header information as necessary. If security services were requested (conditional flag AUTH is true), apply per-message security services. Send the PDU.

The conditional flags and data items set in the run-time system (with prefix SND_) provide the appropriate input for generating the PDU data. Note that actions within the same execution step that started this activity may have assigned values to the SND_* variables which have to be taken by this instance of the activity.

After sending a response PDU, the RT_OUT_FRAG pointer is incremented accordingly, to point to the remaining data in the transmit queue.

Note: The SEND_PKT activity may be invoked simultaneously by several orthogonal states (WORKING, CONTROL, CANCEL, and so on). The run-time system must catch these send requests, buffer these and the associated data, and perform the sends in sequential order.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Activity: STOP_ORPHAN

Description: Tell the stub that the client orphaned the call and allow the manager routine to run down gracefully. If it is executing (that is, it is still receiving for a pipe), cancel it. Otherwise, discard the input and/or output data. If possible (not required), ensure that neither **response** nor **fault** PDU is returned for the orphaned call.

Chart: CO_SERVER

Activity: VERIFY_AUTH

Description: Verify the authentication trailer of PDU and decrypt message if necessary.

This activity takes as input values the PDU header field **auth_proto** and the authentication verifier.

Depending on the result of the verification, the activity VERIFY_AUTH generates either the event AUTHENTICATED (success) or DENIED (authentication failure).

The algorithm applied to this activity is dependent on the security service in use (determined by RT_AUTH_SPEC). The general evaluation steps for authentication service rpc_c_authn_dce_secret are as follows (for more details see Section 13):

- Check the protection level applied to the PDU (parameter in RT_AUTH_VERIFIER) against the protection level for the call (negotiated security context). If matching, proceed with verification, otherwise raise DENIED.

  Note that **bind** and **alter_context** PDUs are used for negotiating the security context. Therefore, the protection level will not be verified for these PDUs; this verification takes only place for actual call PDUs.

- Decrypt the cyphertext portion of the verifier and verify PDUs integrity. If discrepancies are found, raise DENIED, otherwise raise AUTHENTICATED and proceed (if privacy protected).

- If privacy protection is requested, decrypt PDU body data.

Note: The VERIFY_AUTH activity may be invoked simultaneously by several orthogonal states (WORKING, CONTROL and CANCEL). VERIFY_AUTH must not generate the event AUTHENTICATED unless the entire requested authentication processing is completed. If VERIFY_AUTH detects an authentication failure and generates the event DENIED, the protocol machine rejects the RPC call and no further processing is required.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.4.2 CO_SERVER States

The CO_SERVER statechart defines the following states:

Chart: CO_SERVER

State: ASSOCIATION

Description: Main state for active association.

Chart: CO_SERVER

State: ASSOC_OPEN

Description: Association available for call.

Reactions	
Trigger	**Action**
RCV_ALTER_CONTEXT	DO_ALTER_CONTEXT_RESP
RESOURCES_SCARCE	SND_REPLY_TYPE:=SHUTDOWN; st!(SEND_PKT)
DENIED[ALTER_CONTEXT_PDU]	SND_AUTH_VALUE_SUB_TYPE:= CONST_SUB_TYPE_INVALID_CHECKSUM; SND_REPLY_TYPE:=ALTER_CONTEXT_RESP; st!(SEND_PKT)

Chart: CO_SERVER

State: ASSOC_REQ

Description: Wait for decision on whether to accept association.

Reactions	
Trigger	**Action**
ACCEPT_BIND[not GROUP_EXISTS]	tr!(WAIT_FOR_GROUP)
entering	RT_GROUP_ID:=PDU_ASSOC_GROUP_ID
ASSOC_REJECT	SND_REJECT_REASON:=RT_REJECT_REASON; SND_REPLY_TYPE:=BIND_NAK; st!(SEND_PKT)

Chart: CO_SERVER

State: AUTHENTICATION

Description: Process authentication verification.

Activities Throughout:
VERIFY_AUTH

Chart: CO_SERVER

State: CALL

Description: Processing a remote procedure call request.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

State: CANCEL

Description: Processing of client requests to terminate call in progress.

The reaction within this state senses the termination of the CANCEL_NOTIFY_APP activity as cancel acknowledgement from the server manager routine. The manager routine also sets the RETURN_PENDING_CANCEL flag appropriately.

Reactions	
Trigger	**Action**

sp(CANCEL_NOTIFY_APP)	IF
	  RETURN_PENDING_CANCEL
	THEN
	  tr!(SND_PENDING_CANCEL)
	END IF

Chart: CO_SERVER

State: CAN_IDLE

Description: Waits for cancel requests.

Reactions	
Trigger	**Action**

exiting	IF
	  AUTH
	THEN
	  RT_AUTH_VERIFIER_CAN:=PDU_AUTH_VERIFIER;
	  RT_AUTH_LENGTH_CAN:=PDU_AUTH_SPEC
	END IF

Chart: CO_SERVER

State: CLOSED

Description: Association waiting to receive bind request.

Reactions	
Trigger	**Action**
en(CLOSED)	fs!(WAIT_FOR_GROUP)
DENIED[BIND_PDU]	SND_REJECT_REASON:= CONST_REASON_INVALID_CHECKSUM; SND_REPLY_TYPE:=BIND_NAK; st!(SEND_PKT)

Chart: CO_SERVER

State: CNTL_IDLE

Description: Waits for incoming control PDUs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Reactions	
Trigger	**Action**
en(CNTL_IDLE)	fs!(ORPHANED_PDU); fs!(BIND_PDU); fs!(ALTER_CONTEXT_PDU)
RECEIVE_PDU[PDU_TYPE=ORPHANED and VALID_PDU_HEADER]	tr!(ORPHANED_PDU)
exiting	IF AUTH THEN RT_AUTH_VERIFIER_CNTL:= PDU_AUTH_VERIFIER; RT_AUTH_LENGTH_CNTL:= PDU_AUTH_SPEC END IF
RECEIVE_PDU[PDU_TYPE=BIND and VALID_PDU_HEADER]	tr!(BIND_PDU)
RECEIVE_PDU[PDU_TYPE=ALTER_CONTEXT and VALID_PDU_HEADER]	tr!(ALTER_CONTEXT_PDU)
RECEIVE_PDU[CNTL_PDU and not VALID_FRAG_SIZE]	RCV_FRAG_SIZE_TOO_LARGE
RECEIVE_PDU[CNTL_PDU and VALID_PDU_HEADER]	RCV_CNTL_PDU

Chart: CO_SERVER

State: CONTROL

Description: Processing received control PDUs.

Chart: CO_SERVER

State: CO_SERVER

Description: Main state for call and association; created by CONNECT_ESTABLISHED event.

The CO_SERVER state is created when a connection to a server is established, using the primary or secondary address for an association group.

Chart: CO_SERVER

State: DATA

Description: Processing RPC call data.

Chart: CO_SERVER

State: FAULT

Description: Handles processing faults and sends **fault** PDU.

Reactions	
Trigger	**Action**
en(FAULT)	DEALLOC_REQ

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

State: INDICATION

Description: Handles incoming RPC request fragments.

Reactions	
Trigger	**Action**
en(INDICATION) [not LAST_IN_FRAG]	DO_IN_PKT; st!(HANDLE_IN_FRAG)
en(INDICATION) [LAST_IN_FRAG]	DO_IN_PKT; st!(HANDLE_IN_FRAG); RCV_LAST_IN_FRAG
RECEIVE_PDU[PDU_TYPE=REQUEST and not VALID_FRAG_SIZE]	RCV_FRAG_SIZE_TOO_LARGE
RECEIVE_PDU[PDU_TYPE=REQUEST and VALID_PDU_HEADER and not AUTH]	DO_REQ; RCV_FRAG_PDU
RECEIVE_PDU[PDU_TYPE=REQUEST and VALID_PDU_HEADER and AUTH]	DO_REQ; RCV_FRAG_PDU_A

Chart: CO_SERVER

State: INIT

Description: Initial RPC call state. Waits for request from client.

Reactions	
Trigger	**Action**
en(INIT)	fs!(MAYBE); fs!(AUTH); fs!(LAST_IN_FRAG); SND_CANCEL_COUNT:=0; fs!(SND_PENDING_CANCEL)
RECEIVE_PDU[PDU_TYPE=REQUEST and not VALID_FRAG_SIZE]	RCV_FRAG_SIZE_TOO_LARGE
tm(en(INIT), TIMEOUT_SERVER_DISCONNECT)	SND_REPLY_TYPE:=SHUTDOWN; st!(SEND_PKT)
exiting	IF PDU_MAYBE THEN tr!(MAYBE) END IF
exiting	IF PDU_PENDING_CANCEL THEN tr!(SND_PENDING_CANCEL) END IF
exiting	IF PDU_AUTH_SPEC/=0 THEN tr!(AUTH) END IF
exiting	SETUP_CALL

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_SERVER

State: PROCESS_REQ

Description: Promotes completely received request to manager routine

Reactions	
Trigger	**Action**
entering	RT_OUT_PARAMS:=NULL

Chart: CO_SERVER

State: REPLIED

Description: Terminal state for calls.

Reactions	
Trigger	**Action**
en(REPLIED)	DEALLOC_REQ

Chart: CO_SERVER

State: REPLYING

Description: Handles fragmented reply to client.

Reactions	
Trigger	**Action**
en(REPLYING)	fs!(SND_FIRST_FRAG)

Chart: CO_SERVER

State: WORKING

Description: Main working state for call instance.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.4.3 CO_SERVER Events

The CO_SERVER statechart defines the following events:

Chart: CO_SERVER

Event: ABORT_ASSOC_REQ

Description: Abrupt termination of the association requested. Generated externally.

Chart: CO_SERVER

Event: ABORT_CONNECTION

Description: Signal transport to abort connection. Generated internally.

Chart: CO_SERVER

Event: ACCEPT_BIND

Description: Generated by the ACCEPT_ASSOC_POLICY activity.

The local policy permits establishment of the requested association. The mechanism for deciding whether to accept or reject a bind request is implementation and policy-dependent.

Chart: CO_SERVER

Event: ADD_ALTER_CONTEXT

Description: Update the set of presentation contexts for this association.

Select the set of matching presentations contexts based on the received presentation context list (**p_context_elem** field) of the **alter_context** PDU and the contexts supported by the server. Generate the structure **p_result_list** to be sent to the client in the **alter_context_response** PDU.

Chart: CO_SERVER

Event: ADD_PRES_CONTEXT

Description: Update the presentation context set for this association.

Select the set of matching presentations contexts based on the received presentation context list (**p_context_elem** field) of the **bind** PDU and the contexts supported by the server. Generate the structure **p_result_list** to be sent to the client in the **bind_ack** PDU.

Chart: CO_SERVER

Event: ADD_TO_GROUP

Description: Signal group to add this association.

To avoid race conditions, the ASSOCIATION must lock the group before issuing this event and unlock the group only after the event has been processed by the group machine instance.

Event is generated by CO_SERVER and sensed by CO_SERVER_GROUP.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Event: ASSOC_ACCEPT

Description: The server accepted the association.

Definition:
```
ACCEPT_BIND[GROUP_EXISTS] or
 [GROUP_EXISTS and WAIT_FOR_GROUP]
```

Chart: CO_SERVER

Event: ASSOC_REJECT

Description: Generated by the ACCEPT_ASSOC_POLICY activity.

The local policy rejected the request for a new association. The mechanism for deciding whether to accept or reject a bind request is implementation and policy-dependent.

Chart: CO_SERVER

Event: AUTHENTICATED

Description: Authentication processing completed successfully.

Chart: CO_SERVER

Event: CANCEL_CALL

Description: Generate local cancel request for the call currently using the association.

Chart: CO_SERVER

Event: CLOSE

Description: Compound events to terminate association.

Definition:
```
NO_CONNECTION or ABORT_ASSOC_REQ
```

Chart: CO_SERVER

Event: COMPLETE

Description: RPC call completed (with success or fault).

Definition:
```
DEALLOC_REQ or DENIED[not ORPHANED_PDU] or
 [AUTH and TICKET_EXP] or
 RCV_FRAG_SIZE_TOO_LARGE
```

Chart: CO_SERVER

Event: CONNECT_ESTABLISHED

Description: A connection to server has been established. Generated externally by transport.

The address used to establish the connection may be either the primary or, if one exists, secondary address for the server.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Event: DEALLOC_REQ

Description: Call completed or failed. Service provider requests the deallocation of assoc.

Chart: CO_SERVER

Event: DENIED

Description: Authentication failure detected.

The VERIFY_AUTH activity generates this event if either the integrity check failed or the requested protection level for authentication services does not match (not for **bind** or **alter_context** PDUs).

Chart: CO_SERVER

Event: LAST_OUT_PKT

Description: Statechart internal event: last fragment of fragmented response.

Definition: `[TRANSMIT_RESP and LAST_OUT_FRAG]`

Chart: CO_SERVER

Event: MARK_ASSOC

Description: Mark association with termination status and related information.

Chart: CO_SERVER

Event: NEXT_OUT_PKT

Description: Statechart internal event: intermediate fragment of fragmented response.

Definition: `[TRANSMIT_RESP and not LAST_OUT_FRAG]`

Chart: CO_SERVER

Event: NO_CONNECTION

Description: Notification that the underlying connection terminated. Generated externally.

Chart: CO_SERVER

Event: PROCESSING_FAULT

Description: Execution of procedure failed. Returned from called procedure (stub).

Chart: CO_SERVER

Event: PROCESSING_FDNE

Description: Stub (manager routine) or run-time system rejected RPC request.

The call did not execute.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Event: PROC_FAULT

Description: Execution of call failed.

Definition: `PROCESSING_FAULT or PROCESSING_FDNE`

Chart: CO_SERVER

Event: PROC_RESPONSE

Description: Call returned from called procedure (server manager routine).

This event indicates that the called application procedure is ready to respond to the RPC request and has provided **out** parameter data in the RT_OUT_PARAMS queue. The processing of the application procedure may not have been completed and more **out** parameter data may to be queued (sensed by the TRANSMIT_RESP and LAST_OUT_FRAG condition flags).

Chart: CO_SERVER

Event: RCV_ALTER_CONTEXT

Description: Received valid **alter_context** PDU. Generated in CNTL_CALL action.

Chart: CO_SERVER

Event: RCV_BIND

Description: Received valid **bind** PDU on this association's transport connection.

Generated in CNTL_CALL action.

Chart: CO_SERVER

Event: RCV_CAN_PDU

Description: Received **cancel** PDU with valid header.

Definition:
```
RECEIVE_PDU[PDU_TYPE=CANCEL and
 VALID_PDU_HEADER and in(DATA) and not
 in(REPLYING) and not in(REPLIED)]
```

Chart: CO_SERVER

Event: RCV_CNTL_PDU

Description: Received one of the control PDUs with valid header.

Chart: CO_SERVER

Event: RCV_FRAG_PDU

Description: Received PDU for non-authenticated fragmented requests with valid header.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Event: RCV_FRAG_PDU_A

Description: Received PDU for authenticated fragmented request with valid header.

Chart: CO_SERVER

Event: RCV_FRAG_SIZE_TOO_LARGE

Description: The received PDU exceeded the maximum allowed fragment size.

Chart: CO_SERVER

Event: RCV_LAST_IN_FRAG

Description: Received last fragment **request** PDU. Signal completion to stub.

The last fragment of a multi-fragmented request or a single packet request was received. RCV_LAST_IN_FRAG signals that the complete request data is available to the stub for unmarshalling, and it transfers the control from the run-time system to the stub for processing the RPC request.

Chart: CO_SERVER

Event: RCV_REQ_PDU

Description: Received **request** PDU (first packet for fragmented requests) with valid header.

Definition:
```
RECEIVE_PDU[PDU_TYPE=REQUEST and
 in(ASSOC_OPEN) and PDU_FIRST_FRAG and
 VALID_VERSION]
```

Chart: CO_SERVER

Event: RECEIVE_PDU

Description: Received a PDU from client.

Chart: CO_SERVER

Event: REMOVE_FROM_GROUP

Description: Signal association group to remove this association.

To avoid race conditions, the ASSOCIATION must lock the group before issuing this event and unlock the group only after the event has been processed by the group machine instance.

Event is generated by CO_SERVER and sensed by CO_SERVER_GROUP.

Chart: CO_SERVER

Event: RESOURCES_SCARCE

Description: Request to reclaim resources. Externally generated.

Resource management is implementation-specific. This event is generated by the implementation resource management policy when it is necessary to reclaim idle associations. It is recommended that at least one idle association per client-server pair be maintained for better performance. This may be tuned for different style applications.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Event: SEND_RESPONSE

Description: Called procedure provided **out** parameters to be sent.

Definition: `PROC_RESPONSE[not MAYBE]`

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.4.4 CO_SERVER Actions

The CO_SERVER statechart defines the following actions:

Chart: CO_SERVER

Action: CLEAN_UP

Description: Termination actions.

Definition:

```
MARK_ASSOC;
IF
   in(ASSOC_OPEN)
THEN
   REMOVE_FROM_GROUP;
   CANCEL_CALL
END IF;
WHEN
   ABORT_ASSOC_REQ
THEN
   ABORT_CONNECTION
END WHEN
```

Chart: CO_SERVER

Action: CNTL_CALL

Description: Reactions on received control PDUs. Generate respective RCV_* events.

Definition:

```
IF
   ORPHANED_PDU
THEN
   st!(STOP_ORPHAN);
   DEALLOC_REQ
END IF;
IF
   BIND_PDU
THEN
   RCV_BIND
END IF;
IF
   ALTER_CONTEXT_PDU
THEN
   RCV_ALTER_CONTEXT
END IF
```

Chart: CO_SERVER

Action: DO_ALTER_CONTEXT_RESP

Description: Process the alter context negotiation request and send response back.

Note that the activities ADD_ALTER_CONTEXT and SEND_PKT must be synchronised to assure that the **alter_context_resp** PDU contains the negotiated context.

Definition:

```
RT_CLIENT_PRES_CONTEXT_LIST:=PDU_P_CONT_LIST;
RT_IF_ID:=PDU_IF_ID;
RT_IF_VERSION:=PDU_IF_VERSION;
ADD_ALTER_CONTEXT;
SND_REPLY_TYPE:=ALTER_CONTEXT_RESP;
st!(SEND_PKT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Action: DO_BIND_ACK

Description: Signal CO_SERVER_GROUP to add group and send a **bind_ack** PDU.

Definition:

```
RT_IF_ID:=PDU_IF_ID;
RT_IF_VERSION:=PDU_IF_VERSION;
IF
    PDU_MAX_XMIT_FRAG_SIZE>RT_MAX_RCV_FRAG_SIZE
    or PDU_MAX_XMIT_FRAG_SIZE=0 and
    RT_MAX_RCV_FRAG_SIZE<CONST_MUST_RCV_FRAG_SIZE
THEN
   SND_MAX_RCV_FRAG_SIZE:=RT_MAX_RCV_FRAG_SIZE
ELSE
   IF
      PDU_MAX_XMIT_FRAG_SIZE=0
   THEN
      SND_MAX_RCV_FRAG_SIZE:=CONST_MUST_RCV_FRAG_SIZE
   ELSE
      SND_MAX_RCV_FRAG_SIZE:-PDU_MAX_XMIT_FRAG_SIZE
   END IF
END IF;
IF
    PDU_MAX_RCV_FRAG_SIZE>RT_MAX_XMIT_FRAG_SIZE
    or PDU_MAX_RCV_FRAG_SIZE=0 and
    RT_MAX_XMIT_FRAG_SIZE<CONST_MUST_RCV_FRAG_SIZE
THEN
   SND_MAX_XMIT_FRAG_SIZE:=RT_MAX_XMIT_FRAG_SIZE
ELSE
   IF
      PDU_MAX_RCV_FRAG_SIZE=0
   THEN
      SND_MAX_XMIT_FRAG_SIZE:=CONST_MUST_RCV_FRAG_SIZE
   ELSE
      SND_MAX_XMIT_FRAG_SIZE:=PDU_MAX_RCV_FRAG_SIZE
   END IF
END IF;
SND_GROUP_FIELD:=RT_GROUP_ID;
SND_SEC_ADDR:=RT_SECONDARY_ADDRESS;
RT_CLIENT_PRES_CONTEXT_LIST:=PDU_P_CONT_LIST;
ADD_TO_GROUP;
ADD_PRES_CONTEXT;
SND_REPLY_TYPE:=BIND_ACK;
st!(SEND_PKT)
```

Chart: CO_SERVER

Action: DO_IN_PKT

Description: Append received **request** PDU body data to internal buffer.

Definition:

```
RT_IN_PARAMS:=RT_IN_PARAMS+RT_BODY
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_SERVER

Action: DO_REPLY

Description: Send last **out** frag to requesting client.

Definition:
```
fs!(TRANSMIT_RESP);
IF
   LAST_OUT_FRAG
THEN
   tr!(SND_LAST_FRAG)
ELSE
   fs!(SND_LAST_FRAG)
END IF;
SND_OUT_PARAMS:=RT_OUT_FRAG;
SND_REPLY_TYPE:=RESPONSE;
st!(SEND_PKT)
```

Chart: CO_SERVER

Action: DO_REQ

Description: Evaluate **request** PDU header and signal allocation request.

Definition:
```
RT_PRES_CONTEXT_ID:=PDU_P_CONT_ID;
RT_BODY:=PDU_BODY;
IF
   PDU_AUTH_SPEC/=0
THEN
   RT_AUTH_VERIFIER_CALL:=PDU_AUTH_VERIFIER;
   RT_AUTH_LENGTH_CALL:=PDU_AUTH_SPEC
END IF;
IF
   PDU_LAST_FRAG
THEN
   tr!(LAST_IN_FRAG)
END IF
```

Chart: CO_SERVER

Action: ERROR

Description: Determine the type of failure.

Definition:
```
WHEN
   PROCESSING_FDNE
THEN
   tr!(SND_DID_NOT_EXECUTE)
ELSE
   fs!(SND_DID_NOT_EXECUTE)
END WHEN;
FAULT_CALL
```

Chart: CO_SERVER

Action: FAULT_CALL

Description: Send **fault** PDU.

Definition:
```
IF
   not MAYBE
THEN
   SND_REPLY_TYPE:=FAULT;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
   st!(SEND_PKT)
END IF;
IF
   in(PROCESS_REQ) or in(REPLYING) or
    in(REPLIED)
THEN
   st!(ABORT_SEND)
ELSE
   st!(ABORT_RECEIVE)
END IF
```

Chart: CO_SERVER

Action: FINAL

Description: Perform final actions for RPC call.

Definition:

```
WHEN
   DENIED[not BIND_PDU and not
    ALTER_CONTEXT_PDU]
THEN
   SND_OUT_PARAMS:=CONST_NCA_S_INVALID_CHKSUM;
   FAULT_CALL;
   DEALLOC_REQ
END WHEN;
WHEN
   RCV_FRAG_SIZE_TOO_LARGE
THEN
   SND_OUT_PARAMS:=CONST_NCA_S_PROTO_ERROR;
   FAULT_CALL;
   DEALLOC_REQ
END WHEN
```

Chart: CO_SERVER

Action: FIRST_REPLY

Description: Initialise and send first **response** PDU.

Definition:

```
fs!(TRANSMIT_RESP);
IF
   LAST_OUT_FRAG
THEN
   tr!(SND_LAST_FRAG)
ELSE
   fs!(SND_LAST_FRAG)
END IF;
tr!(SND_FIRST_FRAG);
SND_PRES_CONTEXT_ID:=RT_PRES_CONTEXT_ID;
SND_CALL_ID:=RT_CALL_ID;
RT_OUT_FRAG:=RT_OUT_PARAMS;
SND_OUT_PARAMS:=RT_OUT_PARAMS;
SND_REPLY_TYPE:=RESPONSE;
st!(SEND_PKT)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Action: PROCESS_CAN

Description: Process cancel request (signal manager routine).

Definition:
```
SND_CANCEL_COUNT:=SND_CANCEL_COUNT+1;
st!(CANCEL_NOTIFY_APP)
```

Chart: CO_SERVER

Action: SETUP_CALL

Description: Set up call data at first call's **request** PDU.

Definition:
```
RT_CALL_ID:=PDU_CALL_ID;
IF
   PDU_OBJ_UUID
THEN
   RT_OBJ_ID:=PDU_OBJ_ID
ELSE
   RT_OBJ_ID:=NULL
ENDIF;
RT_OP_NUM:=PDU_OP_NUM
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.4.5 CO_SERVER Conditions

The CO_SERVER statechart defines the following conditions:

Chart: CO_SERVER
Condition: ALTER_CONTEXT_PDU
Description: Statechart internal flag: received PDU type **alter_context**.

Chart: CO_SERVER
Condition: AUTH
Description: Statechart internal flag: false if PDU field **auth_length** = 0; true otherwise.

Chart: CO_SERVER
Condition: BIND_PDU
Description: Statechart internal flag: received PDU type **bind**.

Chart: CO_SERVER
Condition: CNTL_PDU
Description: Statechart internal flag: to be received control PDUs.
Definition:
```
PDU_TYPE=ORPHANED or PDU_TYPE=BIND or
 PDU_TYPE=ALTER_CONTEXT
```

Chart: CO_SERVER
Condition: GROUP_EXISTS
Description: The group exists.
Definition:
```
in(CO_SERVER_GROUP:CO_SERVER_GROUP)
```

Chart: CO_SERVER
Condition: LAST_IN_FRAG
Description: Statechart internal flag: last **in** fragment or non-frag **in** packet received.

Chart: CO_SERVER
Condition: LAST_OUT_FRAG
Description: Statechart internal flag: last **out** fragment or non-frag **out** packet ready to send.

This flag is set by the run-time system if the transmit queue contains the last fragment (see also Subclause 9.3 on page 304).

Chart: CO_SERVER
Condition: MAYBE
Description: Statechart internal flag: maybe call.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart:	CO_SERVER
Condition:	ORPHANED_PDU
Description:	Statechart internal flag: received PDU type **orphaned**.

Chart:	CO_SERVER
Condition:	PDU_FIRST_FRAG
Description:	Header flag PFC_FIRST_FRAG.

Chart:	CO_SERVER
Condition:	PDU_LAST_FRAG
Description:	Header flag PFC_LAST_FRAG.

Chart:	CO_SERVER
Condition:	PDU_MAYBE
Description:	Header flag PFC_MAYBE.

Chart:	CO_SERVER
Condition:	PDU_OBJECT_UUID
Description:	Status if optional **object** field is present in received PDU (header flag PFC_OBJECT_UUID is set).

Chart:	CO_SERVER
Condition:	PDU_PENDING_CANCEL
Description:	Header flag PFC_PENDING_CANCEL in received **request** PDU.

Chart:	CO_SERVER
Condition:	RETURN_PENDING_CANCEL
Description:	Cancel pending state returned from stub after processing the cancel request.

Chart:	CO_SERVER
Condition:	SND_DID_NOT_EXECUTE
Description:	Statechart internal flag: send **fault** PDU with PFC_DID_NOT_EXECUTE header flag set.

Chart:	CO_SERVER
Condition:	SND_FIRST_FRAG
Description:	Statechart internal flag: send first **out** fragment.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Condition: SND_LAST_FRAG

Description: Statechart internal flag: header flag PFC_LAST_FRAG for PDU to be sent.

Chart: CO_SERVER

Condition: SND_PENDING_CANCEL

Description: Cancel pending state for current call at server.

The cancel pending state is set by the server manager routine via the CANCEL_NOTIFY_APP activity.

Chart: CO_SERVER

Condition: TICKET_EXP

Description: Statechart internal flag: ticket expired.

Definition:
```
SYS_TIME>GRACE_PERIOD+PDU_EXP_TIME
```

Chart: CO_SERVER

Condition: TRANSMIT_RESP

Description: One or more fragments queued for transmission of response data.

This flag indicates that one or more response fragment(s) are queued in a run-time internal buffer and ready to be transmitted.

The run-time system internally sets this flag (true) after the stub initially provides data in the transmit queue, sufficient for at least the first PDU fragment to be transmitted. The protocol machine resets this flag if it has detected and taken an event for sending the next fragment in the queue.

The run-time system sets this flag again after completion of a SEND_PKT if the transmit queue contains enough data for the next PDU fragment to be transmitted.

Chart: CO_SERVER

Condition: VALID_FRAG_SIZE

Description: Evaluation whether received PDU exceeds size limit.

Definition:
```
not PDU_TYPE=BIND and
 PDU_FRAG_LENGTH<=RT_MAX_RCV_FRAG_SIZE or
 PDU_TYPE=BIND and
 PDU_FRAG_LENGTH<=CONST_MUST_RCV_FRAG_SIZE
```

Chart: CO_SERVER

Condition: VALID_PDU_HEADER

Description: Pre-evaluation of PDU header (before authentication processing).

Definition:
```
PDU_CALL_ID=RT_CALL_ID and VALID_VERSION
 and VALID_FRAG_SIZE
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Condition: VALID_VERSION

Description: *Evaluation of protocol version.*

Definition:
```
PDU_VERSION_NUM=CO_VERSION_NUM_V20 and
 PDU_VERSION_NUM_MINOR<=CO_VERSION_NUM_V20_MINOR
```

Chart: CO_SERVER

Condition: WAIT_FOR_GROUP

Description: *Association waits for group creation before opening.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.4.6 CO_SERVER Data-Items

The CO_SERVER statechart defines the following data items:

Chart: CO_SERVER
Data Item: ALTER_CONTEXT
Description: Constant: PDU type **alter_context**.
Definition: 14

Chart: CO_SERVER
Data Item: ALTER_CONTEXT_RESP
Description: Constant: PDU type **alter_context_resp**.
Definition: 15

Chart: CO_SERVER
Data Item: BIND
Description: Constant: PDU type **bind**.
Definition: 11

Chart: CO_SERVER
Data Item: BIND_ACK
Description: Constant: PDU type **bind_ack**.
Definition: 12

Chart: CO_SERVER
Data Item: BIND_NAK
Description: Constant: PDU type **bind_nak**.
Definition: 13

Chart: CO_SERVER
Data Item: CANCEL
Description: Constant: PDU type **cancel**.
Definition: 18

Chart: CO_SERVER
Data Item: CONST_MUST_RCV_FRAG_SIZE
Description: Constant: MustRecvFragSize value, indicating the lower bound of the fragment size.
Definition: 1432

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Data Item: CONST_NCA_S_INVALID_CHKSUM

Description: Constant: reject status code nca_s_invalid_chksum.

Chart: CO_SERVER

Data Item: CONST_NCA_S_PROTO_ERROR

Description: Constant: reject status code nca_s_proto_error.

Chart: CO_SERVER

Data Item: CONST_REASON_INVALID_CHECKSUM

Description: The value indicating a security integrity failure.

This value is the invalid_checksum member of the enumerated type **p_reject_reason_t** (see Section 12). This is transmitted in the **provider_reject_reason** field of the **bind_nak** PDU.

Chart: CO_SERVER

Data Item: CONST_SUB_TYPE_INVALID_CHECKSUM

Description: Value indicating a security integrity failure (invalid checksum).

The value dce_c_cn_dce_sub_type_invalid_checksum, which is encoded in the **sub_type** field of the **auth_value** member of the authentication verifier. (See Section 13.)

Definition: 2

Chart: CO_SERVER

Data Item: CO_VERSION_NUM_V20

Description: Constant: RPC protocol version 2.0 major version number.

Definition: 5

Chart: CO_SERVER

Data Item: CO_VERSION_NUM_V20_MINOR

Description: Constant: RPC protocol minor version number.

Chart: CO_SERVER

Data Item: FAULT

Description: Constant: PDU type fault.

Definition: 3

Chart: CO_SERVER

Data Item: GRACE_PERIOD

Description: Grace period on server after ticket expiration (implementation-specific).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER
Data Item: ORPHANED
Description: Constant: PDU type **orphaned**.
Definition: 19

Chart: CO_SERVER
Data Item: PDU_ASSOC_GROUP_ID
Description: The **assoc_group_id** field from the received **bind** PDU.

Chart: CO_SERVER
Data Item: PDU_AUTH_SPEC
Description: PDU header field **auth_length**.

Chart: CO_SERVER
Data Item: PDU_AUTH_VERIFIER
Description: PDU trailer: authentication verifier (authentication protocol-specific).

Chart: CO_SERVER
Data Item: PDU_BODY
Description: Array of PDU body data.

Chart: CO_SERVER
Data Item: PDU_CALL_ID
Description: PDU header field **call_id**.

Chart: CO_SERVER
Data Item: PDU_EXP_TIME
Description: Ticket expiration time transmitted in the authentication verifier.

Chart: CO_SERVER
Data Item: PDU_FRAG_LENGTH
Description: PDU header field **frag_length**.

Chart: CO_SERVER
Data Item: PDU_IF_ID
Description: PDU header field: interface identifier, encoded in the **p_context_elem** field of **bind** and **alter_context** PDUs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Chart: CO_SERVER
Data Item: PDU_IF_VERSION
Description: PDU header field: interface version, encoded in the **p_context_elem** field of **bind** and **alter_context** PDUs.

Chart: CO_SERVER
Data Item: PDU_MAX_RCV_FRAG_SIZE
Description: PDU header field **max_recv_frag**.

Chart: CO_SERVER
Data Item: PDU_MAX_XMIT_FRAG_SIZE
Description: PDU header field **max_xmit_frag**.

Chart: CO_SERVER
Data Item: PDU_OBJ_ID
Description: PDU header field **object**.

Chart: CO_SERVER
Data Item: PDU_OP_NUM
Description: PDU header field **opnum**.

Chart: CO_SERVER
Data Item: PDU_P_CONT_ID
Description: PDU header field **p_cont_id**.

Chart: CO_SERVER
Data Item: PDU_P_CONT_LIST
Description: PDU header field **p_cont_elem** in **bind** and **alter_context** PDUs.

Chart: CO_SERVER
Data Item: PDU_TYPE
Description: PDU header field **PTYPE**.

Chart: CO_SERVER
Data Item: PDU_VERSION_NUM
Description: PDU header field **rpc_vers**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER
Data Item: PDU_VERSION_NUM_MINOR
Description: PDU header field **rpc_vers_minor**.

Chart: CO_SERVER
Data Item: REQUEST
Description: Constant: PDU type **request**.
Definition: 0

Chart: CO_SERVER
Data Item: RESPONSE
Description: Constant: PDU type **response**.
Definition: 2

Chart: CO_SERVER
Data Item: RT_AUTH_LENGTH_CALL
Description: Statechart internal: **auth_length** field received in CALL state.

Chart: CO_SERVER
Data Item: RT_AUTH_LENGTH_CAN
Description: Statechart internal: **auth_length** field received in CANCEL state.

Chart: CO_SERVER
Data Item: RT_AUTH_LENGTH_CNTL
Description: Statechart internal: **auth_length** field received in CONTROL state.

Chart: CO_SERVER
Data Item: RT_AUTH_VERIFIER_CALL
Description: Received authentication trailer (verifier) for **request** PDU.

Chart: CO_SERVER
Data Item: RT_AUTH_VERIFIER_CAN
Description: Received authentication trailer (verifier) for **cancel** PDU.

Chart: CO_SERVER
Data Item: RT_AUTH_VERIFIER_CNTL
Description: Received authentication trailer (verifier) for **control** PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER
Data Item: RT_BODY
Description: Statechart internal: temporarily buffered **request** PDU body data.

Chart: CO_SERVER
Data Item: RT_CALL_ID
Description: Statechart internal: call identifier of current RPC call.

Chart: CO_SERVER
Data Item: RT_CLIENT_PRES_CONTEXT_LIST
Description: Statechart internal: presentation context as represented by the client.

Chart: CO_SERVER
Data Item: RT_GROUP_ID
Description: The identifier of the association group of which this association is a member.

Chart: CO_SERVER
Data Item: RT_IF_ID
Description: Statechart internal: received interface UUID.

Chart: CO_SERVER
Data Item: RT_IF_VERSION
Description: Statechart internal: received interface version number.

Chart: CO_SERVER
Data Item: RT_IN_PARAMS
Description: Statechart internal: buffered array of reassembled input data.

Chart: CO_SERVER
Data Item: RT_MAX_RCV_FRAG_SIZE
Description: Maximum size of a fragment the receiver is able to handle.

The minimum value of this fragment size is determined by the architected value MustRcvFragSize (refer to Section 12).

Implementations may support larger fragment sizes that are subject to negotiation with the client. This value is set internally by run-time implementations.

Chart: CO_SERVER
Data Item: RT_MAX_XMIT_FRAG_SIZE
Description: Maximum size of a fragment the sender is able to handle.

The minimum value of this fragment size is determined by the architected value MustRcvFragSize (refer to Section 12).

Implementations may support larger fragment sizes that are subject to negotiation with the client. This value is set internally by run-time implementations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER

Data Item: RT_OBJ_ID

Description: Statechart internal: buffered object UUID of RPC call.

Chart: CO_SERVER

Data Item: RT_OP_NUM

Description: Statechart internal: buffered operation number of RPC call.

Chart: CO_SERVER

Data Item: RT_OUT_FRAG

Description: Statechart internal pointer to data to be sent in next **response** PDU.

The SEND_PKT activity increments this pointer after a **response** PDU is sent.

Chart: CO_SERVER

Data Item: RT_OUT_PARAMS

Description: Buffered array of unfragmented output data.

RT_OUT_PARAMS is the queue of transmit data provided by the stub. A possible segmentation of this queue is not equivalent to the sizes of PDU fragments sent by the run-time system (SEND_PKT) activity.

The RT_OUT_FRAG variable is a pointer data type that points to the to be transmitted data fragment within this RT_IN_PARAMS queue.

Chart: CO_SERVER

Data Item: RT_PRES_CONTEXT_ID

Description: Statechart internal: presentation context identifier of current call.

Chart: CO_SERVER

Data Item: RT_REJECT_REASON

Description: The reason the bind request was rejected.

The RPC run-time system sets this value according to the detected error (see also the **p_reject_reason_t** type definition in Section 12).

Chart: CO_SERVER

Data Item: RT_SECONDARY_ADDRESS

Description: Secondary address for this server.

Chart: CO_SERVER

Data Item: SHUTDOWN

Description: Constant: PDU type **shutdown**.

Definition: 17

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER
Data Item: SND_AUTH_VALUE_SUB_TYPE
Description: The value of the **sub_type** field of the **auth_value** member of the authentication verifier sent in an **alter_context_resp** PDU. (See Section 13.)

Chart: CO_SERVER
Data Item: SND_CALL_ID
Description: Call identifier to be sent.

Chart: CO_SERVER
Data Item: SND_CANCEL_COUNT
Description: Counter of received cancel requests for current call.

Chart: CO_SERVER
Data Item: SND_GROUP_FIELD
Description: The **assoc_group_id** field of a **bind_ack** PDU.

Chart: CO_SERVER
Data Item: SND_MAX_RCV_FRAG_SIZE
Description: **max_recv_frag** header value to be sent.

Chart: CO_SERVER
Data Item: SND_MAX_XMIT_FRAG_SIZE
Description: **max_xmit_frag** header field to be sent.

Chart: CO_SERVER
Data Item: SND_OUT_PARAMS
Description: PDU body data promoted to SEND_PKT activity.

Chart: CO_SERVER
Data Item: SND_PRES_CONTEXT_ID
Description: Presentation context identifier to be sent.

Chart: CO_SERVER
Data Item: SND_REJECT_REASON
Description: The value sent for the reject reason in a **bind_nak** PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER
Data Item: SND_REPLY_TYPE
Description: PDU type to be sent.

Chart: CO_SERVER
Data Item: SND_SEC_ADDR
Description: The **sec_addr** field of a **bind_ack** PDU to be sent.

Chart: CO_SERVER
Data Item: SYS_TIME
Description: Secure reference time of local system.

Chart: CO_SERVER
Data Item: TIMEOUT_SERVER_DISCONNECT
Description: Timeout value: DefaultServerDisconnectTimer.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.5 CO_SERVER_GROUP Machine

Figure 11-5 shows the CO_SERVER_GROUP machine statechart.

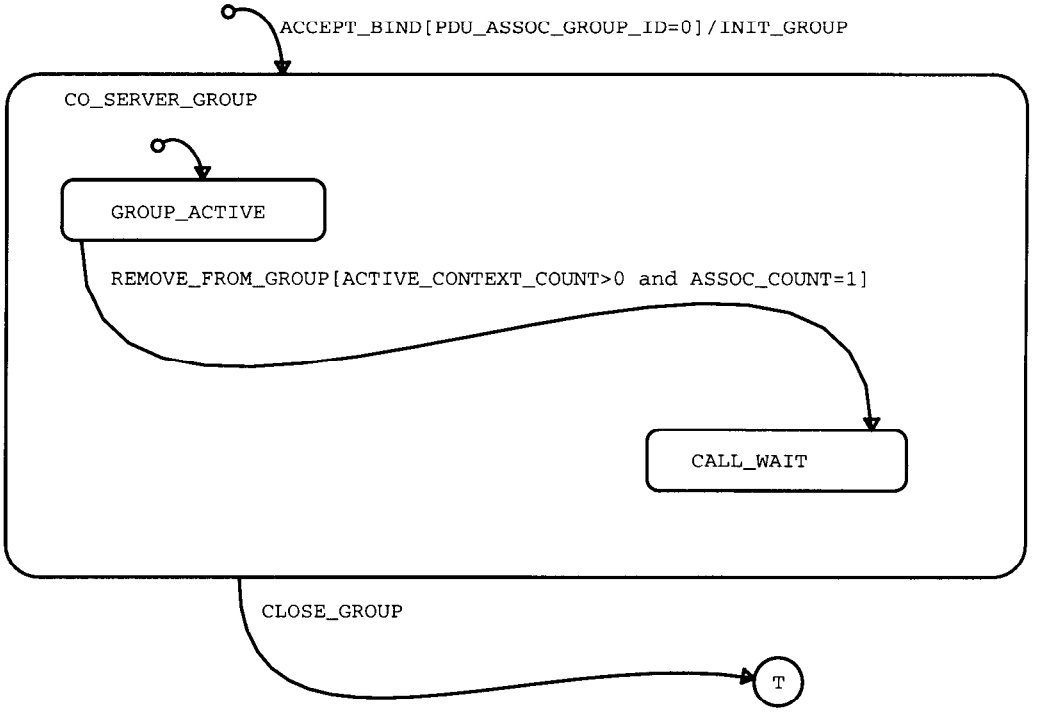


Figure 11-5 CO_SERVER_GROUP Statechart

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.1 CO_SERVER_GROUP States

The CO_SERVER_GROUP statechart defines the following states:

Chart: CO_SERVER_GROUP

State: CALL_WAIT

Description: Wait for calls to complete before running down context handles.

Allows the server an opportunity to complete before attempting context rundown.

Reactions	
Trigger	**Action**
exiting	IF ACTIVE_CONTEXT_COUNT>0 THEN RUNDOWN_CONTEXT_HANDLES END IF
CONTEXT_ACTIVE@T{ ACTIVE_CONTEXT_COUNT:= ACTIVE_CONTEXT_COUNT+1	
CONTEXT_INACTIVE	ACTIVE_CONTEXT_COUNT:= ACTIVE_CONTEXT_COUNT-1

Chart: CO_SERVER_GROUP

State: CO_SERVER_GROUP

Description: Main state for a server association group. Created by ACCEPT_BIND event.

Note that once the CO_SERVER_GROUP is terminated, the group ID associated with this group is no longer valid. Receipt of a PDU containing a PDU_ASSOC_GROUP_ID which does not match the group ID of any group is a client protocol error.

Chart: CO_SERVER_GROUP

State: GROUP_ACTIVE

Description: Group contains associations not in CLOSED state.

Reactions	
Trigger	**Action**
ADD_TO_GROUP	ASSOC_COUNT:=ASSOC_COUNT+1
REMOVE_FROM_GROUP[ASSOC_COUNT>1]	ASSOC_COUNT:=ASSOC_COUNT-1
CONTEXT_ACTIVE	ACTIVE_CONTEXT_COUNT:= ACTIVE_CONTEXT_COUNT+1
CONTEXT_INACTIVE	ACTIVE_CONTEXT_COUNT:= ACTIVE_CONTEXT_COUNT-1

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

11.5.2 CO_SERVER_GROUP Events

The CO_SERVER_GROUP statechart defines the following events:

Chart: CO_SERVER_GROUP

Event: ACCEPT_BIND

Description: Externally generated. Server accepts association. Same as in CO_SERVER.

Chart: CO_SERVER_GROUP

Event: ADD_TO_GROUP

Description: Signal group to add this association.

To avoid race conditions, the ASSOCIATION must lock the group before issuing this event and unlock the group only after the event has been processed by the group machine instance.

Event is generated by CO_SERVER and sensed by CO_SERVER_GROUP.

Chart: CO_SERVER_GROUP

Event: CLOSE_GROUP

Description: Close the group.

Definition:
```
NO_CALLS[in(CALL_WAIT)] or
 REMOVE_FROM_GROUP[ACTIVE_CONTEXT_COUNT=0
 and ASSOC_COUNT=1]
```

Chart: CO_SERVER_GROUP

Event: CONTEXT_ACTIVE

Description: A context handle was activated. Generated by the server stub.

The stub must generate this event for each context handle which makes a transition from inactive to active. To avoid a race condition which could result from multiple simultaneous events, the stub must lock the group before generating the CONTEXT_ACTIVE event and release the lock only after the event has been processed by the group machine.

Chart: CO_SERVER_GROUP

Event: CONTEXT_INACTIVE

Description: Context handle deactivated. Generated by the server stub.

The stub generates this event for each context handle which makes a transition from active to inactive. To avoid a race condition which could result from multiple simultaneous events, the stub must lock the group before generating the CONTEXT_INACTIVE event and release the lock only after the event has been processed by the group machine.

Chart: CO_SERVER_GROUP

Event: NO_CALLS

Description: All calls using this association group have completed. Generated externally.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Chart: CO_SERVER_GROUP

Event: REMOVE_FROM_GROUP

Description: Signal association group to remove this association.

To avoid race conditions, the ASSOCIATION must lock the group before issuing this event and unlock the group only after the event has been processed by the group machine instance.

Event is generated by CO_SERVER and sensed by CO_SERVER_GROUP.

Chart: CO_SERVER_GROUP

Event: RUNDOWN_CONTEXT_HANDLES

Description: Signal stub to rundown all active context handles for this group.

The stub manages context handles and may associate them with rundown routines. An instance of the CO_SERVER_GROUP signals the stub to rundown any active context handles that were associated with this group.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.3 CO_SERVER_GROUP Actions

The CO_SERVER_GROUP statechart defines the following actions:

Chart: CO_SERVER_GROUP

Action: INIT_GROUP

Description: Initialise state for group and generate value for RT_GROUP_ID.

Definition:
```
ASSOC_COUNT:=0;
ACTIVE_CONTEXT_COUNT:=0
```

11.5.4 CO_SERVER_GROUP Data-Items

The CO_SERVER_GROUP statechart defines the following data items:

Chart: CO_SERVER_GROUP

Data Item: ACTIVE_CONTEXT_COUNT

Description: Number of active context handles for this group. Internal variable.

Chart: CO_SERVER_GROUP

Data Item: ASSOC_COUNT

Description: Number of associations in group. Internal variable.

Chart: CO_SERVER_GROUP

Data Item: PDU_ASSOC_GROUP_ID

Description: The group id field from the received **bind** PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

This page intentionally left blank.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 12. RPC PDU Encodings

This Section specifies the encodings of the Protocol Data Units (PDUs) used by the connectionless and connection-oriented RPC protocols. The first subclause provides common information for the two protocols. Subsequent subclauses provide connectionless and connection-oriented protocol-specific information.

Table 12-1 lists the PDUs that are specified.

12.1 Generic PDU Structure

An RPC PDU contains up to three parts:

- A PDU header that contains protocol control information. A header is present in all PDUs.
- A PDU body that contains data. For example, the body of a **request** or **response** PDU contains data representing the input or output parameters for an operation. A body is present only in some types of PDUs.
- An authentication verifier that contains data specific to an authentication protocol. For example, an authentication protocol may ensure the integrity of a packet via inclusion of an encrypted checksum in the authentication verifier. The presence of an authentication verifier depends on the PDU type and whether authentication is being used.

Table 12-1 RPC Protocol Data Units

PDU Type	Protocol	Type Value
request	CO/CL	0
ping	CL	1
response	CO/CL	2
fault	CO/CL	3
working	CL	4
nocall	CL	5
reject	CL	6
ack	CL	7
cl_cancel	CL	8
fack	CL	9
cancel_ack	CL	10
bind	CO	11
bind_ack	CO	12
bind_nak	CO	13
alter_context	CO	14
alter_context_resp	CO	15
shutdown	CO	17
co_cancel	CO	18
orphaned	CO	19

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.2 Encoding Conventions

The encodings are provided here as IDL data type declarations. The actual declarations of PDU data types are implementation-dependent.

The run-time system treats PDU headers as byte streams that are encoded according to the Network Data Representation (NDR, see Section 14) encoding rules. In PDUs containing stub data, the stub data may be encoded according to any of the negotiated presentation syntaxes. PDU-specific header fields identify the stub data presentation syntax. The run-time system identifies the presentation syntax from the PDU-specific header fields and interprets the stub data accordingly.

In order to encode and decode the header fields, implementations must support the following subset of NDR:

- integers: 1, 2 and 4 octet unsigned
- octet string
- record constructor for the preceding types
- conformant array constructor.

Implementations must follow the rule that, beginning at the first octet of the PDU (or the first octet of the optional data field), each field is encoded in the order specified without any additional information such as type, length or padding.

12.3 Alignment

For all the PDUs, the scalar header fields are aligned **[0 MOD min(8,sizeof(field))]**. For PDUs that contain stub data, the header is padded, if necessary, with zeros to an integral multiple of 8 octets in length. This allows the stub data to assume it will always begin with **0 MOD 8** alignment. Padding, where necessary, is included in the type declarations. The **align(n)** function, which appears in some connection-oriented PDU declarations, identifies a function that returns the number of padding octets to force alignment to a natural multiple of **n**. Implementations should ensure that each message in memory begins on an 8 octet alignment boundary in order to preserve the natural alignment.

Refer to Section 13 for alignment requirements for the optional authentication verifiers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.4 Common Fields

Header encodings differ between connectionless and connection-oriented PDUs. However, certain fields use common sets of values with a consistent interpretation across the two protocols. These values are specified in the following subclauses.

12.4.1 PDU Types

Both connectionless and connection-oriented PDU headers contain a 1-byte field that identifies the PDU type. The values are shown in the column marked **Type Value** in Table 12-1 on page 465.

12.4.2 Protocol Version Numbers

Both connectionless and connection-oriented PDU headers contain version number fields that indicate the RPC protocol version. The connectionless headers contain a single version field, while the connection-oriented headers contain both a major and a minor version field. Connectionless and connection-oriented protocol version numbers vary independently, so that a given version number is not necessarily unique to one of the protocols.

The version numbers for the protocols specified in this Section are as follows:

- The major version number for the connection-oriented protocol is 5.
- The minor version number for the connection-oriented protocol is 0 (zero).
- The version number for the connectionless protocol is 4.

12.4.3 Data Representation Format Labels

Both connectionless and connection-oriented PDU headers contain an NDR data representation format label that identifies the formats used by the sender of a PDU to represent data in the PDU header, and when the transfer syntax is NDR, in the PDU body. (The representation of data in an authentication verifier is determined by the authentication protocol.) Section 14 describes the NDR data representation format label.

As defined by NDR, the format label consists of 4 bytes, although the fourth byte is currently unused. Only the first 3 bytes appear in RPC connectionless PDU headers. Connection-oriented PDUs include space for all four bytes of the format label.

NDR defines only one bit layout for the format label itself, so its format is the same in all PDUs.

12.4.4 Reject Status Codes

Both **reject** and connection-oriented **fault** PDUs contain a 32-bit field that indicates a server's reason for rejecting an RPC call request. This field is encoded as the body data of the **reject** PDU and as the **status** field of the connection-oriented **fault** PDU header. Table E-1 on page 549 lists the possible values of this field.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.5 Connectionless RPC PDUs

The RPC run-time system uses the connectionless PDUs for the client/server communications that are required by each remote procedure call over a connectionless transport. The following subclauses specify the encoding of each of the connectionless PDUs.

12.5.1 Connectionless PDU Structure

Connectionless PDUs consist of a header, body data and an optional authentication verifier. The PDU header has a fixed length of 80 bytes. The encoding is described in Subclause 12.5.2. The PDU body can be defined as an array of the IDL **byte** type. The length of the PDU body is specified in the PDU header.

The authentication verifier can be defined as an array of the IDL **byte** type. The length of the authentication verifier depends on the authentication protocol specified in the PDU header. An authentication verifier is present in a PDU only if the **auth_proto** header field is not 0 (zero).

The maximum size of an RPC connectionless PDU is the fixed header length (80 bytes) plus the maximum body length plus the length of the authentication verifier, which is determined by the authentication protocol. In practice, however, an implementation of RPC over a particular connectionless transport can impose a lower maximum. For example, if an implementation is layered on a connectionless transport whose maximum PDU size is 1 KB, then the maximum PDU size is 1 KB.

If the input data in a request or the output data in a response exceed the maximum PDU body size, the RPC connectionless protocols fragment the request or response into several PDUs, called fragments. The fragment number can be reused for a given call if the fragment number space is exhausted. If fragment numbers wrap around and are reused, the implementation must assure that these are unambiguous (for example, the first 50% of fragments must have been acknowledged).

Requests that are broadcast must fit in one PDU, and on some connectionless transports they may be subject to further size limitations.

12.5.2 Header Encoding

The connectionless header can be defined as a structure in Interface Definition Language (IDL):

```
typedef struct {
    unsigned small rpc_vers = 4; /* RPC protocol major version (4 LSB only)*/
    unsigned small ptype;       /* Packet type (5 LSB only) */
    unsigned small flags1;      /* Packet flags */
    unsigned small flags2;      /* Packet flags */
    byte           drep[3];     /* Data representation format label */
    unsigned small serial_hi;   /* High byte of serial number */
    uuid_t         object;      /* Object identifier */
    uuid_t         if_id;       /* Interface identifier */
    uuid_t         act_id;      /* Activity identifier */
    unsigned long  server_boot;/* Server boot time */
    unsigned long  if_vers;     /* Interface version */
    unsigned long  seqnum;      /* Sequence number */
    unsigned short opnum;       /* Operation number */
    unsigned short ihint;       /* Interface hint */
    unsigned short ahint;       /* Activity hint */
    unsigned short len;         /* Length of packet body */
    unsigned short fragnum;     /* Fragment number */
    unsigned small auth_proto; /* Authentication protocol identifier*/
    unsigned small serial_lo;   /* Low byte of serial number */
} dc_rpc_cl_pkt_hdr_t;
```

The bit layout of each field in a PDU header is determined by the following:

- The IDL type of the field, as shown in the definition for **dc_rpc_cl_pkt_hdr_t**.
- The NDR representation for that IDL type, as specified in Section 14.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- The NDR character, integer and floating-point formats used by the RPC implementation that is sending the PDU.

12.5.2.1 Protocol Version Number

The protocol version number is a non-negative integer that is encoded in the 4 least significant bits of the **rpc_vers** field. (The remaining bits are reserved.) This number is incremented at each new release of the protocol. The protocol version number allows implementations of several versions of RPC to coexist in a distributed environment. The PDU formats given here are for version 4.

12.5.2.2 PDU Type

The PDU type is a non-negative integer that is encoded in the 5 least significant bits of the **ptype** field. (The remaining bits are reserved.) The values for each type are shown in the column labelled **Type Value** in Table 12-1 on page 465.

12.5.2.3 Flags Fields

Each of the flags fields is an 8-bit integer that is composed of bit flags for use in protocol control.

This International Standard currently specifies the use of 6 bits in the first set of PDU flags. The other bits are either reserved for use by implementations or reserved for future use, as indicated in Table 12-3 on page 470 and Table 12-4 on page 472.

Some flags are meaningful only in PDUs that are sent from the client to the server. These flags are ignored in PDUs that are sent from the server to the client.

Table 12-3 on page 470 lists the bit flags in the first set of PDU flags.

Table 12-2 The First Set of PDU Flags

PDU Flag	Hex Value	Meaning
reserved_01	01	Reserved for use by implementations.
lastfrag	02	Meaningful in either direction. If set, the PDU is the last fragment of a multi-PDU transmission.
frag	04	Meaningful in either direction. If set, the PDU is a fragment of a multi-PDU transmission.
nofack	08	Meaningful for fragments sent in either direction. If set, the receiver is not requested to send a **fack** PDU for the fragment. Otherwise, if not set, the receiver acknowledges the received PDU with a **fack** PDU. Note that both client and server may send **fack** PDUs independent of the status of this flag.
maybe	10	Meaningful only from client to server. If set, the PDU is for a maybe request.
idempotent	20	Meaningful only from client to server. If set, the PDU is for an idempotent request.
broadcast	40	Meaningful only from client to server. If set, the PDU is for a broadcast request.
reserved_80	80	Reserved for use by implementations.

Table 12-4 on page 472 lists the bit flags in the second set of PDU flags.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 12-3 Second Set of PDU Flags

PDU Flag	Hex Value	Meaning
reserved_01	01	Reserved for use by implementations.
cancel_pending	02	Cancel pending at the call end.
reserved_04	04	Reserved for future use. Must be set to 0.
reserved_08	08	Reserved for future use. Must be set to 0.
reserved_10	10	Reserved for future use. Must be set to 0.
reserved_20	20	Reserved for future use. Must be set to 0.
reserved_40	40	Reserved for future use. Must be set to 0.
reserved_80	80	Reserved for future use. Must be set to 0.

12.5.2.4 Data Representation Format Label

The data representation format label is described in Section 14. As defined by NDR, the format label consists of 4 bytes, although the fourth byte is currently unused. Only the first 3 bytes appear in connectionless PDUs.

12.5.2.5 Serial Number

The serial number is a 16-bit non-negative integer that identifies a transmission of a fragment.

The serial number is set to 0 (zero) when a remote procedure call is initiated, and is incremented after each time a fragment is sent or resent on behalf of that call.

In a **request** or **response** PDU that is part of a multi-PDU transmission, the serial number provides information to help determine the temporal order of fragment transmissions and retransmissions. In other types of PDUs, the serial number is meaningless.

The 2 bytes of the serial number do not occupy contiguous positions in the PDU header. The most significant byte follows the data representation format label. The least significant byte follows the authentication protocol number.

12.5.2.6 Object Identifier

The object identifier is a UUID that uniquely identifies the object on which a remote procedure call is operating. If the call does not operate on an object, this field contains the nil UUID.

The server uses the object UUID, the interface UUID, the interface version number, and the operation number to select the operation to execute on the client's behalf.

12.5.2.7 Interface Identifier

The interface identifier is a UUID that uniquely identifies the interface being called.

The server uses the object UUID, the interface UUID, the interface version number, and the operation number to select the operation to execute on the client's behalf.

12.5.2.8 Activity Identifier

The activity identifier is a UUID that uniquely identifies the client activity that is making a remote procedure call. The server can use the activity UUID as a communications key between it and the client.

12.5.2.9 Server Boot Time

The server boot time is a 32-bit non-negative integer that indicates the time at which the current instantiation of a server was booted; that is, the time at which the process in which the server is running was created, not the time at which the server host was booted. Server boot time is represented as time in seconds since 1 January 1970 and must increase with each boot of the server.

A server passes its boot time in all the PDUs that it sends to a client. The client passes back this value in all the PDUs that it subsequently sends to the same server. When a client sends its first PDU to a server, it does not know the server boot time, and it passes a value of 0 (zero). The server boot time field enables the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RPC connectionless protocols to prevent nonidempotent operations from being executed more than once in the face of a server crash.

12.5.2.10 Interface Version

The interface version is a 32 bit non-negative integer that identifies the version number of the interface being called. This field allows servers to implement several versions of one interface.

The server uses the object UUID, the interface UUID, the interface version number, and the operation number to select the operation to execute on the client's behalf.

12.5.2.11 Sequence Number

The sequence number is a 32-bit non-negative integer that identifies the remote procedure call that an activity is making.

Each remote procedure call invoked by an activity has a unique sequence number that is assigned when the call is initiated. All RPC connectionless PDUs sent on behalf of that particular call have the same sequence number, whether the PDUs are from client to server or from server to client.

When an activity initiates a new remote procedure call, it increases the sequence number, so that each subsequent call has a larger sequence number. Together, the activity UUID and the sequence number uniquely identify a remote procedure call.

12.5.2.12 Operation Number

The operation number is a 16-bit non-negative integer that identifies a particular operation within the interface being called.

12.5.2.13 Interface Hint

The interface hint is a 16-bit non-negative integer. Although an implementation can use this field for any purpose, its intended use is to allow a server to optimise lookups of information about interfaces.

12.5.2.14 Activity Hint

The activity hint is a 16-bit non-negative integer. Although an implementation can use this field for any purpose, its intended use is to allow a server to optimise lookups of information about the state of its communications with an activity.

12.5.2.15 PDU Body Length

The PDU body length is a 16-bit non-negative integer that indicates the length in bytes of the PDU body. The maximum PDU body size is 65528 bytes. The alignment requirements for the PDU header (see Subclause 12.3 on page 466) also apply to the PDU body data.

12.5.2.16 Fragment Number

The fragment number is a 16-bit non-negative integer that identifies a PDU within a multi-PDU transmission.

In a **request** or **response** PDU that is part of a multi-PDU transmission, the fragment number indicates the fragment that is being sent. The fragment number is incremented for each fragment. The first fragment is fragment 0, the second is fragment 1, and so on.

In a **fack** PDU and a **nocall** PDU with a body, the fragment number indicates the fragments that have been received in order, as follows:

- If fragment 0 through fragment **n** have been received, and fragment **n**+1 has not been received, the fragment number field should be **n**.
- If fragment 0 has not been received, the fragment number field should contain the hexadecimal value FFFF.

In effect, fragments received out of order are not acknowledged.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.5.2.17 Authentication Protocol Identifier

The authentication protocol identifier is an 8-bit non-negative integer that identifies an authentication protocol.

Table 12-4 lists the possible values for the authentication protocol identifier field and briefly describes the authentication protocol identified by each value. All other values are reserved for future use.

Table 12-4 Authentication Protocol Identifiers

Identifier	Protocol Used
0	None
1	OSF DCE Private Key Authentication

12.5.3 Connectionless PDU Definitions

The following subclauses describe the contents and use of each of the connectionless PDUs.

12.5.3.1 The ack PDU

A client sends an **ack** PDU after it has received a response to an **at-most-once** request. An **ack** PDU explicitly acknowledges that the client has received the response; it tells the server to cease resending the response and discard the response PDU. (A client can also implicitly acknowledge receipt of a response by sending a new request to the server.)

An **ack** PDU contains no body data.

12.5.3.2 The cancel_ack PDU

A server sends a **cancel_ack** PDU after it has received a **cancel** PDU. A **cancel_ack** PDU acknowledges that the server has cancelled or orphaned a remote call or indicates that the server is not accepting cancels.

A **cancel_ack** PDUs can optionally have a body. A **cancel_ack** PDU without a body acknowledges orphaning of a call, whereas a **cancel_ack** PDU with a body acknowledges cancellation of a call. Orphaned calls do not perform any further processing. Canceled calls transparently deliver a notification to the server manager routine without altering the run-time system state of the call. The run-time system's processing of a cancelled call continues uninterrupted.

When a **cancel_ack** PDU has a body, its format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned32  vers = 0; /* cancel-request body format version */
    unsigned32  cancel_id;/* id of a cancel-request event being ack'd */
    boolean     server_is_accepting; /* server accepting cancels ? */
} rpc_dg_cancel_ack_body_t;
```

- An NDR **unsigned long** that indicates the version number of the **cancel_ack** PDU body. This is independent of the protocol version number contained in the PDU header. This specification is for version 0.
- An NDR **unsigned long** that identifies the cancel request event that is being acknowledged.
- An NDR **boolean** that indicates whether the server is accepting the cancel request. TRUE means that it is accepting.

The version number for the format of the **cancel_ack** body is the first byte of the body. This version number changes independently of the protocol version number in the PDU header.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.5.3.3 The cancel PDU

A client sends a **cancel** PDU when it has incurred a cancel fault. A **cancel** PDU tells the server to cancel or orphan a remote operation. Canceled calls transparently deliver a notification to the server manager routine without altering the run-time system state of the call. The run-time system's processing of a cancelled call continues uninterrupted.

The **cancel** PDU body format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned32  vers = 0; /* cancel body format version */
    unsigned32  cancel_id;/* id of a cancel-request event */
} rpc_dg_cancel_body_t;
```

- An NDR **unsigned long** that indicates the version number of the **cancel** PDU body. This is independent of the protocol version number contained in the PDU header. This specification is for version 0.
- An NDR **unsigned long** that identifies a cancel request event.

The version number for the format of the **cancel** body is the first byte of the body. This version number changes independently of the protocol version number in the PDU header.

12.5.3.4 The fack PDU

Both clients and servers send **fack** PDUs.

A **client** sends a **fack** PDU after it has received a fragment of a multi-PDU response. A **fack** PDU explicitly acknowledges that the client has received the fragment; it may tell the sender to stop sending for a while.

A **server** sends a **fack** PDU after it has received a fragment of a multi-PDU request. A **fack** PDU explicitly acknowledges that the server has received the fragment; it may tell the sender to stop sending for a while.

The **nofack** bit flag in a request or response fragment header can be used to control the sending of **fack** PDUs by the receiver as follows:

- The receiver of a fragment must send an **fack** if the fragment's **nofack** flag is not set.
- The receiver of a fragment may choose not to send a **fack** if the fragment's **nofack** flag is set.

A **fack** PDU may contain a body with data that can be used by the sender of a request or response to increase the efficiency of a multi-PDU transmission. The contents of a **fack** PDU body are specified, but use of the data is left to implementations. Implementations must be able to receive **fack** PDUs without bodies.

The first byte of a **fack** PDU body is a version number for the format of the **fack** body. This version number can change independently of the protocol version number in the PDU header.

At version 0, the **fack** PDU body format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned8 vers = 0;        /* Fack packet body version */
    u_char pad1;
    unsigned16 window_size;    /* Sender's receive window size (in pkts) */
    unsigned32 max_tsdu;       /* largest local TPDU size */
    unsigned32 max_path_tpdu; /* largest TPDU not fragmented */
    unsigned16 serial_num;     /* serial # of packet that induced this fack */
    unsigned16 selack_len;     /* number of elements in the selack array    */
    unsigned32 selack[1];      /* variable number of 32 bit selective ack   */
                               /* bit masks.                                */
} rpc_dg_fack_body_t;
```

- An NDR **unsigned short** integer indicates the version number of the **fack** PDU body.
- An uninitialised pad byte.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- An NDR **unsigned short** integer indicates the size in KB of the receive window at the sender of the **fack**. The window size indicates how much additional data the sender of the **fack** is prepared to receive. The window size field can be used by senders and receivers to coordinate their rates of transmission and reception.
- An NDR **unsigned long** integer indicates the size in bytes of the largest Transport Protocol Data Unit (TPDU) that can be passed through the local transport service interface at the sender of the **fack**.
- An NDR **unsigned long** integer indicates the size in bytes of the largest TPDU that can be sent from the sender of the **fack** to the receiver of the **fack** without being fragmented, as estimated by the sender of the **fack**.
- An NDR **unsigned short** integer indicates the serial number of the fragment transmission that the sender of the **fack** is acknowledging. See Subclause 9.5.5 on page 310 for the semantics of serial numbers.
- An NDR **unsigned short** integer that indicates the number of elements in the array of selective acknowledgement bit masks. Selective acknowledgement bit masks identify any fragments that the sender of the **fack** has received out of order; that is, any fragments whose fragment numbers are greater than the fragment number in the header of the **fack**.
- An array of NDR **unsigned long** integers that function as bit masks, indicating any fragments that the sender of the **fack** has received out of order.

Suppose that the sender of a **fack** has received fragments 0 through **n**, but not fragment **n**+1 of a multi-PDU transmission; **n** is therefore the fragment number in the header of the **fack**. Let **m** be an index for a bit mask in the bit mask array, where a value of **m**=0 indicates the first bit mask in the array. Let **b** be an index for a bit in a bit mask, where a value of **b**=0 indicates the least significant bit and a value of **b**=31 indicates the most significant bit. The value of bit **b** in mask **m** indicates whether the sender of the **fack** has received the fragment with fragment number **n**+32**m**+**b**+1. A value of 1 indicates that the fragment has been received, and a value of 0 indicates that the fragment has not been received.

The first bit in the first mask (**m**=0 and **b**=0) must always have a value of 0, since fragment **n**+1 is the first missing fragment.

A **fack** body contains only as many selective acknowledgement bit masks as are necessary to acknowledge fragments received out of order. The last bit mask in the **fack** body must always have at least one non-zero bit.

The information in selective acknowledgement bit masks is intended to tell the sender of a multi-PDU transmission which fragments it may need to resend.

12.5.3.5 The fault PDU

A server sends a **fault** PDU if an operation incurs a fault while executing on the server side.

The **fault** PDU body format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned32 st; /* status code */
} rpc_dg_fault_body_t;
```

The body of a **fault** PDU contains a status code that indicates the fault that a requested operation incurred. The status code is represented by an NDR **unsigned long**. The fault status values and the corresponding application level **fault_status** parameter values are listed in Annex E.

12.5.3.6 The nocall PDU

A server sends a **nocall** PDU in reply to a **ping** PDU. This reply indicates that the server is not currently processing the client's call. The server may have never received the request, or some of the request fragments may have been lost and need to be retransmitted.

A **nocall** PDU can optionally carry a body whose format is the same as the optional **fack** PDU body. (See Subclause 12.5.3.4 on page 473.) If the server wants to indicate that the call in question is queued but not

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

yet processed, it sets the receive window size to zero to indicate to the client that it need not resend the request.

12.5.3.7 The ping PDU

A client sends a **ping** PDU when it wants to inquire about an outstanding request.

A **ping** PDU contains no body data.

12.5.3.8 The reject PDU

A server sends a **reject** PDU if an RPC request is rejected. The body of a **reject** PDU contains a status code indicating why a callee is rejecting a **request** PDU from a caller. The body format is the same as that of the **fault PDU**. (See Subclause 12.6.4.7 on page 488.) The status code is represented by an NDR **unsigned long**. Reject status codes are listed Annex E.

Note that reject status codes map to application level **comm_status** parameter values. This mapping is given in Annex E.

12.5.3.9 The request PDU

A client sends a **request** PDU when it wants to execute a remote operation. In a multi-PDU request, the request consists of a series of **request** PDUs with the same sequence number and monotonically increasing fragment numbers. The body of a **request** PDU contains data that represents input parameters for the operation.

A **request** PDU may have one of the following types:

idempotent The request is for an idempotent operation. An idempotent request has the **idempotent** bit flag set.

broadcast The request is to be broadcast to all hosts on the local network. A broadcast request has the **idempotent** and **broadcast** bit flags set. Broadcast calls are never authenticated. The request must be sent in one PDU.

maybe The client does not expect a response. The protocols do not guarantee that the server will receive the request. A maybe request has the **idempotent** and **maybe** bit flags set.

broadcast/maybe The request is to be broadcast to all hosts on the local network and the client does not expect a response. A broadcast/maybe request has the **idempotent**, **broadcast** and **maybe** bit flags set. The request must be sent in one PDU.

at-most-once The request is for an operation that cannot be executed more than once. An at-most-once request is the default; none of the **idempotent**, **broadcast** or **maybe** bit flags is set.

The body of a **request** PDU consists of an NDR representation of the input parameters for the request.

12.5.3.10 The response PDU

A server sends a **response** PDU if an operation invoked by an idempotent, broadcast or at-most-once request executes successfully. Servers do not send responses for maybe or broadcast/maybe requests. A multi-PDU response consists of a series of **response** PDUs with the same sequence number and monotonically increasing fragment numbers.

The body of a **response** PDU consists of the NDR representation of the output parameters for the response.

12.5.3.11 The working PDU

A server sends a **working** PDU in reply to a **ping** PDU. This reply indicates that the server is processing the client's call.

A **working** PDU contains no body data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.6 Connection-oriented RPC PDUs

The RPC run-time system uses the connection-oriented PDUs for the client/server communications required by each remote procedure call over a connection-oriented transport. This subclause specifies the encoding of each of the connection-oriented PDUs listed in Table 12-1 on page 465.

The client and server **CALL** protocol machines communicate using a set of call PDUs, and the client and server **ASSOCIATION** protocol machines communicate using a set of association PDUs. (The **ASSOCIATION GROUP** state machines are purely local to the client and server and exchange no PDUs.) The PDUs in each group are:

Association	**bind**
	bind_ack
	bind_nak
	alter_context
	alter_context_response
Call	**request**
	response
	fault
	shutdown
	cancel
	orphaned

The association and call PDUs are encoded by the RPC run-time system and delivered to the underlying transport for transmission. This International Standard does not specify encoding of any connection PDUs. In the RPC connection management model, connections are established by the underlying transport. The RPC run-time system assumes that the underlying transport provides certain services (see Section 9), but it does not specify the concrete implementation of these services or any protocol encodings.

12.6.1 Connection-oriented PDU Structure

The connection-oriented PDUs follow the general structure described in Subclause 12.1 on page 465; that is, a header followed by body data and an optional authentication verifier. Connection-oriented PDU headers vary in size; every header includes a set of common header fields, but in some connection-oriented PDUs, this is followed by PDU specific header fields. The authentication verifier may be present in **bind**, **bind_ack**, **alter_context** and **alter_context_response** PDUs; it is never present in **bind_nak** and **shutdown** PDUs; and it is optionally present, depending on security protocol, in the other PDUs.

12.6.2 Fragmentation and Reassembly

RPC request and response service requests may contain arbitrary amounts of stub data. Either of these requests may be fragmented by the RPC run-time system into multiple Transport Service Data Units (TSDUs) and reassembled by the receiving RPC run-time system before or during unmarshalling.

Note: No other RPC service requests shall be fragmented. Others must fit into a single **MustRecvFragSize** fragment.

Each fragment is labelled as such using the **PFC_FIRST_FRAG** and **PFC_LAST_FRAG** flags in the header **pfc_flags** field. If a service request needs only a single fragment, that fragment will have both the **PFC_FIRST_FRAG** and **PFC_LAST_FRAG** flags set to TRUE. Since the connection-oriented transport guarantees sequentiality, the receiver will always receive the fragments in order.

Note: The flags encoding for the connectionless and connection-oriented protocols is different. The connectionless **PF_FRAG** flag is not required in the connection-based encoding, which explicitly labels both the first and last fragment.

This International Standard defines a lower bound on the size of a single fragment (TSDU) that all implementations must be able to receive, **MustRecvFragSize**. A client and server may also negotiate larger fragment sizes as part of the binding operation.

The client determines, and then sends in the **bind** PDU, its desired maximum size for transmitting fragments, and its desired maximum receive fragment size. Similarly, the server determines its desired maximum sizes for transmitting and receiving fragments. Transmit and receive sizes may be different to help preserve buffering. When the server receives the client's values, it sets its operational transmit size to the minimum of the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

client's receive size (from the **bind** PDU) and its own desired transmit size. Then it sets its actual receive size to the minimum of the client's transmit size (from the **bind**) and its own desired receive size. The server then returns its operational values in the **bind_ack** PDU. The client then sets its operational values from the received **bind_ack** PDU. The received transmit size becomes the client's receive size, and the received receive size becomes the client's transmit size.

Either party may use receive buffers larger than negotiated — although this will not provide any advantage — but may not transmit larger fragments than negotiated.

Note: An implementation may ignore the negotiation by always specifying the default encodings of 0 (zero) in the PDUs. This is consistent with the negotiation algorithm described here.

An implementation may wish to adjust its desired fragment sizes to tune them to the most common data links expected.

When receiving a **request** PDU, the **PFC_PENDING_CANCEL** and **PFC_MAYBE** flag values on the **first_frag** are authoritative. When receiving a **response** or **fault** PDU, the **PFC_DID_NOT_EXECUTE** and **PFC_PENDING_CANCEL** flag values and the **cancel_count** and **status** fields of the **last_frag** are authoritative.

12.6.3 Connection-oriented PDU Data Types

The following subclauses provide IDL declarations for header data types. These include both the common header fields and other header fields that appear in various PDUs. These subclauses also discuss the use of several fields and data types by the connection-oriented RPC protocol.

12.6.3.1 Declarations

The following synonyms appear in many of the declarations:

```
typedef     unsigned hyper  u_int64;
typedef     unsigned long   u_int32;
typedef     unsigned short  u_int16;
typedef     unsigned small  u_int8;   /* single octet unsigned int */
```

The common header fields, which appear in all PDU types, are as follows. The comment fields show the exact octet alignment and octet length of each element.

```
/* start 8-octet aligned */
u_int8  rpc_vers = 5;         /* 00:01 RPC version */
u_int8  rpc_vers_minor = 0;   /* 01:01 minor version */
u_int8  PTYPE;                /* 02:01 packet type */
u_int8  pfc_flags;            /* 03:01 flags (see PFC_... ) */
byte    packed_drep[4];    /* 04:04 NDR data representation format label */
u_int16 frag_length;          /* 08:02 total length of fragment */
u_int16 auth_length;          /* 10:02 length of auth_value */
u_int32 call_id;              /* 12:04 call identifier */
```

The NDR data representation format label is discussed in Section 14.

This International Standard defines the following values for the common header **pfc_flags** field:

```
#define PFC_FIRST_FRAG           0x01/* First fragment */
#define PFC_LAST_FRAG            0x02/* Last fragment */
#define PFC_PENDING_CANCEL       0x04/* Cancel was pending at sender */
#define PFC_RESERVED_1           0x08
#define PFC_CONC_MPX             0x10/* supports concurrent multiplexing
                                      * of a single connection. */
#define PFC_DID_NOT_EXECUTE      0x20/* only meaningful on 'fault' packet;
                                      * if true, guaranteed call did not
                                      * execute. */
#define PFC_MAYBE                0x40/* 'maybe' call semantics requested */
#define PFC_OBJECT_UUID          0x80/* if true, a non-nil object UUID
                                      * was specified in the handle, and
                                      * is present in the optional object
                                      * field. If false, the object field
                                      * is omitted. */
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Several elements are used for presentation context identification and negotiation. Local context identifiers are defined as:

```
typedef   u_int16   p_context_id_t;
```

Presentation syntax is identified by:

```
typedef   struct   {
             uuid_t   if_uuid;
             u_int32   if_version;
          } p_syntax_id_t;
```

For abstract syntax, **if_uuid** is set to the interface UUID, and **if_version** is set to the interface version. For transfer syntax, these are set to the UUID and version created for the data representation. The major version is encoded in the 16 least significant bits of **if_version** and the minor version in the 16 most significant bits.

One element in a presentation context list is defined as:

```
typedef   struct {
          p_context_id_t   p_cont_id;
          u_int8           n_transfer_syn;      /* number of items */
          u_int8           reserved;            /* alignment pad, m.b.z. */
          p_syntax_id_t    abstract_syntax;     /* transfer syntax list */
          p_syntax_id_t [size_is(n_transfer_syn)] transfer_syntaxes[];
          } p_cont_elem_t;
```

The whole list is defined as:

```
typedef   struct {
          u_int8          n_context_elem;      /* number of items */
          u_int8          reserved;            /* alignment pad, m.b.z. */
          u_short         reserved2;           /* alignment pad, m.b.z. */
          p_cont_elem_t [size_is(n_cont_elem)] p_cont_elem[];
          } p_cont_list_t;
```

The following declarations are for the results of a presentation context negotiation. Result types are defined as:

```
typedef  short enum {
         acceptance, user_rejection, provider_rejection
         } p_cont_def_result_t;
```

Reasons for rejection of a context element are defined as:

```
typedef   short enum {
          reason_not_specified,
          abstract_syntax_not_supported,
          proposed_transfer_syntaxes_not_supported,
          local_limit_exceeded
          } p_provider_reason_t;
```

The meanings of these rejection reasons are defined in Subclause 2 of the ISO 8823 standard.

A result list returns the results of the context negotiation. A list element is declared as:

```
typedef struct {
        p_cont_def_result_t   result;
        p_provider_reason_t   reason;  /* only relevant if result !=
                                        * acceptance */
        p_syntax_id_t         transfer_syntax;/* tr syntax selected
                                        * 0 if result not
                                        * accepted */
        } p_result_t;
```

The entire list is defined as:

```
/* Same order and number of elements as in bind request */

typedef struct {
        u_int8   n_results;      /* count */
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        u_int8   reserved;       /* alignment pad, m.b.z. */
        u_int16 reserved2;       /* alignment pad, m.b.z. */
        p_result_t [size_is(n_results)] p_results[];
        } p_result_list_t;
```

The protocol version data type is defined as:

```
typedef struct {
        u_int8   major;
        u_int8   minor;
        } version_t;
```

The run-time version data type is synonymous:

```
typedef   version_t   p_rt_version_t;
```

When the protocol negotiation fails, the list of supported protocols is returned as:

```
typedef struct {
        u_int8   n_protocols;   /* count */
        p_rt_version_t  [size_is(n_protocols)] p_protocols[];
        } p_rt_versions_supported_t;
```

The following data structure is used when a bind request returns a secondary address. It holds a string representation of the local port part of the address only. The length includes the C NULL string termination.

```
typedef struct {
        u_int16 length;
        char   [size_is(length)] port_spec; /* port string spec */
        } port_any_t;
```

Reasons for rejection of an association are returned in the **bind_nak** PDU. These are defined as:

```
typedef short enum {
              reason_not_specified,
              temporary_congestion,
              local_limit_exceeded,
              called_presentation_address_unknown,
              protocol_version_not_supported,
              default_context_not_supported,
              user_data_not_readable,
              no_PSAP_available,
              authentication_type_not_recognized,
             invalid_checksum
              } p_reject_reason_t;
```

12.6.3.2 Connection-Oriented Protocol Versions

Each PDU contains the sender's major and minor RPC run-time protocol version numbers. The client's and server's major versions must be equal. Backward compatible changes in the protocol are indicated by higher minor version numbers. Therefore, a server's minor version must be greater than or equal to the client's. However, if the server's minor version exceeds the client's minor version, it must return the client's minor version and restrict its use of the protocol to the minor version specified by the client. A protocol version mismatch causes the **nca_s_rpc_version_mismatch** error status to be returned.

The PDU formats given here are for major version **DC_PROTO_VERS_MAJOR_1**, as defined in Table 12-2 on page 469, minor version 0.

The protocol version is negotiated using the **bind**, **bind_ack** and **bind_nak** messages. For all other messages, the protocol version in the header only serves as a sanity check; if it is incorrect, it indicates a massive error and the connection should be terminated with the error **nca_s_rpc_version_mismatch**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.3.3 The frag_length Field

The **frag_length** field represents the length of the entire PDU, including all of the header, optional header fields, stub body and optional authentication verifier, if applicable.

12.6.3.4 Context Identifiers

Presentation context identifiers are transmitted on each request or response. The client defines the values of the context identifiers, and both the client and server must be able to map between the specific client's context identifier and the selected abstract and transfer syntax, which indicate the interface and data representation.

The client must assign context identifiers that are unique within at least a single association. Context identifiers may also be unique within an association group or across the entire client instantiation. A server must interpret context identifiers relative to each particular association; that is, different associations within the same association group from the same client to the same server may legally use the same context identifier with a different meaning.

12.6.3.5 The call_id Field

Each run-time protocol message contains a **call_id** field. This field is used by the caller to guarantee that it is matching the proper response and request. Otherwise, responses for the wrong call, or orphaned responses from calls that were cancelled, and the cancel timed out, could be confused with current responses. The caller must guarantee that at any time, all **call_id**s for all outstanding messages for the same association group are distinct. The server just returns the **call_id** on the corresponding message.

The **call_id** field is also used to guarantee proper matching of **bind_ack**, **bind_nak** or **alter_context_response** messages in order to guarantee proper behaviour under certain conditions; for example, cancel timeout causes an orphan.

Note: The most straightforward implementation is for each client process to maintain a single **u_int32** sequence number counter to use for the **call_id**. Alternatively, the client may assign a value representing a call data structure, and use that, provided it does sufficient bookkeeping to insure that it cannot be reused if a call is cancelled and times out, at least until the the entire orphaned response is received.

12.6.3.6 The assoc_group_id Field

The client should set the **assoc_group_id** field either to 0 (zero), to indicate a new association group, or to the known value. When the server receives a value of 0, this indicates that the client has requested a new association group, and it assigns a server unique value to the group. This value is returned in the **rpc_bind_ack** PDU.

12.6.3.7 The alloc_hint Field

The **alloc_hint** field may be used by the transmitter as a hint to the receiver, informing it how much buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation; a receiver is required to work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

12.6.3.8 Authentication Data

If the **auth_length** header field value is non-zero, then the message contains optional authentication and/or authorisation data in an authentication verifier. Each message format specifies the location of the verifier, which always follows any stub data, if applicable.

The contents of the authentication verifier are specified in Section 13.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.6.3.9 Optional Connect Reject and Disconnect Data

If the transport supports optional connect reject or disconnect data, the RPC run-time system may transmit additional error information as the optional data. The following data types are used to declare this optional data:

```
typedef  u_int16 rpcrt_reason_code_t;/* 0..65535 */

typedef  struct {
         u_int8    rpc_vers;
         u_int8    rpc_vers_minor
         u_int8    reserved[2];/* must be zero */
         byte      packed_drep[4];
         u_int32   reject_status;
         u_int8    reserved[4];
         } rpcrt_optional_data_t;
```

The format for optional reject data is as follows:

```
typedef struct {
        rpcrt_reason_code_t     reason_code;  /* 0..65535   */
        rpcrt_optional_data_t   rpc_info;     /* may be RPC specific */
        }  rpcconn_reject_optional_data_t;
```

The format for optional disconnect data is as follows:

```
typedef struct {
        rpcrt_reason_code_t     reason_code;  /* 0..65535 */
        rpcrt_optional_data_t   rpc_info;     /* may be RPC-specific */
        } rpcconn_disc_optional_data_t;
```

12.6.4 Connection-oriented PDU Definitions

The following subclauses give IDL declarations and descriptions for each of the the connection-oriented PDUs.

12.6.4.1 The alter_context PDU

The IDL declaration of the **alter_context** PDU is as follows:

```
typedef struct {
      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;          /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;    /* 01:01 minor version */
        u_int8  PTYPE = alter_context; /* 02:01 alter context PDU */
        u_int8  pfc_flags;             /* 03:01 flags */
        byte    packed_drep[4];        /* 04:04 NDR data rep format label*/
        u_int16 frag_length;           /* 08:02 total length of fragment */
        u_int16 auth_length;           /* 10:02 length of auth_value */
        u_int32  call_id;              /* 12:04 call identifier */

      /* end common fields */

        u_int16 max_xmit_frag;         /* ignored */
        u_int16 max_recv_frag;         /* ignored */
        u_int32 assoc_group_id;        /* ignored */

      /* presentation context list */

        p_cont_list_t  p_context_elem; /* variable size */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
auth_trailer_co_t    auth_verifier;

} rpcconn_alter_context_hdr_t;
```

The **alter_context** PDU is used to request additional presentation negotiation for another interface and/or version. The format is identical to the **bind** PDU, except that the value of the **PTYPE** field is set to **alter_context**. The **max_xmit_frag**, **max_recv_frag** and **assoc_group_id** fields are be ignored.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4.2 The alter_context_resp PDU

The IDL declaration of the **alter_context_resp** PDU is as follows:

```
typedef struct {
      /* start 8-octet aligned */

      /* common fields */
       u_int8  rpc_vers = 5;       /* 00:01 RPC version */
       u_int8  rpc_vers_minor = 0; /* 01:01 minor version */
        u_int8  PTYPE = alter_context_response; /* 02:01 alter
                                                   context response PDU */
        u_int8  pfc_flags;          /* 03:01 flags */
        byte    packed_drep[4];     /* 04:04 NDR data rep format label*/
        u_int16 frag_length;        /* 08:02 total length of fragment */
        u_int16 auth_length;        /* 10:02 length of auth_value */
        u_int32  call_id;           /* 12:04 call identifier */

      /* end common fields */

        u_int16 max_xmit_frag;      /* ignored */
        u_int16 max_recv_frag;      /* ignored */
        u_int32 assoc_group_id;     /* ignored */
        port_any_t sec_addr;        /* ignored */

      /* restore 4-octet alignment */

        u_int8 [size_is(align(4))] pad2;

      /* presentation context result list, including hints */

        p_result_list_t     p_result_list;    /* variable size */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier; /* xx:yy */
    } rpcconn_alter_context_response_hdr_t;
```

The **alter_context_response** PDU is used to indicate the server's response to an **alter_context** request. The PDU format is identical to **bind_ack**, except that the value of the **PTYPE** field is set to **alter_context_response**. The **max_xmit_frag**, **max_recv_frag**, **assoc_group_id** and **sec_addr** fields are ignored.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4.3 The bind PDU

The IDL declaration of the **bind** PDU is as follows:

```
/*  bind header */
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;         /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;   /* 01:01 minor version */
        u_int8  PTYPE = bind;         /* 02:01 bind PDU */
        u_int8  pfc_flags;            /* 03:01 flags */
        byte    packed_drep[4];       /* 04:04 NDR data rep format label*/
        u_int16 frag_length;          /* 08:02 total length of fragment */
        u_int16 auth_length;          /* 10:02 length of auth_value */
        u_int32  call_id;             /* 12:04 call identifier */

      /* end common fields */

        u_int16 max_xmit_frag;     /* 16:02 max transmit frag size, bytes */
        u_int16 max_recv_frag;     /* 18:02 max receive  frag size, bytes */
        u_int32 assoc_group_id;    /* 20:04 incarnation of client-server
                                                          * assoc group */
      /* presentation context list */

        p_cont_list_t  p_context_elem; /* variable size */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier;

        } rpcconn_bind_hdr_t;
```

The **bind** PDU is used to initiate the presentation negotiation for the body data, and optionally, authentication. The presentation negotiation follows the model of the OSI presentation layer.

The PDU contains a priority-ordered list of supported presentation syntaxes, both abstract and transfer, and context identifiers (local handles). (This differs from OSI, which does not specify any order for the list.) The abstract and transfer syntaxes are represented as a record of interface UUID and interface version. (These may map one-to-one into OSI object identifiers by providing suitable prefixes and changing the encoding.) Each supported data representation, such as NDR, will be assigned an interface UUID, and will use that UUID as part of its transfer syntax value. Each stub computes its abstract syntax value given its interface UUID and interface version. The transfer syntax value for NDR is defined in Annex I.

The fields **max_xmit_frag** and **max_recv_frag** are used for fragment size negotiation as described in Subclause 12.6.3 on page 477.

The **assoc_group_id** field contains either an association group identifier that was created during a previous bind negotiation or 0 (zero) to indicate a request for a new group.

This PDU shall not exceed the **MustRecvFragSize**, since no size negotiation has yet occurred. If the **p_context_elem** is too long, the leading subset should be transmitted, and additional presentation context negotiation can occur in subsequent **alter_context** PDUs, as needed, after a successful **bind_ack**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.6.4.4 The bind_ack PDU

The IDL declaration of the **bind_ack** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;       /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0; /* 01:01 minor version */
        u_int8  PTYPE = bind_ack;   /* 02:01 bind ack PDU */
        u_int8  pfc_flags;          /* 03:01 flags */
        byte    packed_drep[4];     /* 04:04 NDR data rep format label*/
        u_int16 frag_length;        /* 08:02 total length of fragment */
        u_int16 auth_length;        /* 10:02 length of auth_value */
        u_int32  call_id;           /* 12:04 call identifier */

      /* end common fields */

        u_int16 max_xmit_frag;      /* 16:02 max transmit frag size */
        u_int16 max_recv_frag;      /* 18:02 max receive  frag size */
        u_int32 assoc_group_id;     /* 20:04 returned assoc_group_id */
        port_any_t sec_addr;        /* 24:yy optional secondary address
                                     * for process incarnation; local port
                                     * part of address only */
      /* restore 4-octet alignment */

        u_int8 [size_is(align(4))] pad2;

      /* presentation context result list, including hints */

        p_result_list_t     p_result_list;    /* variable size */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier; /* xx:yy */
    } rpcconn_bind_ack_hdr_t;
```

The **bind_ack** PDU is returned by the server when it accepts a bind request initiated by the client's **bind** PDU. It contains the results of presentation context and fragment size negotiations. It may also contain a new association group identifier if one was requested by the client.

The **max_xmit_frag** and **max_recv_frag** fields contain the maximum transmit and receive fragment sizes as determined by the server in response to the client's desired sizes.

The **p_result_list** contains the results of the presentation context negotiation initiated by the client. It is possible for a **bind_ack** not to contain any mutually supported syntaxes.

If the client requested a new association group, **assoc_group_id** contains the identifier of the new association group created by the server. Otherwise, it contains the identifier of the previously created association group requested by the client.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4.5 The bind_nak PDU

The IDL declaration of the **bind_nak** PDU is as follows:

```
typedef struct {

  /* start 8-octet aligned */

  /* common fields */
    u_int8  rpc_vers = 5;        /* 00:01 RPC version */
    u_int8  rpc_vers_minor = 0; /* 01:01 minor version */
    u_int8  PTYPE = bind_nak;    /* 02:01 bind nak PDU */
    u_int8  pfc_flags;           /* 03:01 flags */
    byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
    u_int16 frag_length;         /* 08:02 total length of fragment */
    u_int16 auth_length;         /* 10:02 length of auth_value */
    u_int32  call_id;            /* 12:04 call identifier */

  /* end common fields */

    p_reject_reason_t provider_reject_reason; /* 16:02 presentation
                                                 context reject */

    p_rt_versions_supported_t versions; /* 18:yy array of protocol
                                         * versions supported */
    } rpcconn_bind_nak_hdr_t;
```

The **bind_nak** PDU is returned by the server when it rejects an association request initiated by the client's **bind** PDU. The **provider_reject_reason** field holds the rejection reason code. When the reject reason is **protocol_version_not_supported**, the **versions** field contains a list of run-time protocol versions supported by the server.

The **bind_nak** PDU never contains an authentication verifier.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.6.4.6 The cancel PDU

The IDL declaration of the **cancel** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5 ;         /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;    /* 01:01 minor version */
        u_int8  PTYPE = co_cancel;     /* 02:01 CO cancel PDU */
        u_int8  pfc_flags;             /* 03:01 flags */
        byte    packed_drep[4];        /* 04:04 NDR data rep format label*/
        u_int16 frag_length;           /* 08:02 total length of fragment */
        u_int16 auth_length;           /* 10:02 length of auth_value */
        u_int32  call_id;              /* 12:04 call identifier */

      /* end common fields */

      /* optional authentication verifier
         * following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier; /* xx:yy */

} rpcconn_cancel_hdr_t;
```

The **cancel** PDU is used to forward a cancel.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.6.4.7 The fault PDU

The IDL declaration of the **fault** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;         /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;   /* 01:01 minor version */
        u_int8  PTYPE = fault;        /* 02:01 fault PDU */
        u_int8  pfc_flags;            /* 03:01 flags */
        byte    packed_drep[4];       /* 04:04 NDR data rep format label*/
        u_int16 frag_length;          /* 08:02 total length of fragment */
        u_int16 auth_length;          /* 10:02 length of auth_value */
        u_int32  call_id;             /* 12:04 call identifier */

      /* end common fields */

      /* needed for request, response, fault */

        u_int32  alloc_hint;      /* 16:04 allocation hint */
        p_context_id_t p_cont_id; /* 20:02 pres context, i.e. data rep */

      /* needed for response or fault */

        u_int8  cancel_count      /* 22:01 received cancel count */
        u_int8  reserved;         /* 23:01 reserved, m.b.z. */

      /* fault code  */

        u_int32  status           /* 24:04 run-time fault code or zero */

      /* always pad to next 8-octet boundary */

        u_int8  reserved2[4];     /* 28:04 reserved padding, m.b.z. */

      /* stub data here, 8-octet aligned
               .
               .
               .                          */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

       auth_trailer_co_t   auth_verifier; /* xx:yy */

} rpcconn_fault_hdr_t;
```

The **fault** PDU is used to indicate either an RPC run-time, RPC stub, or RPC-specific exception to the client. The **p_cont_id** field holds a context identifier that identifies the data representation.

The **alloc_hint** field is optionally used by the client to provide a hint to the receiver of the amount of buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation. The server must work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

The **status** field indicates run-time status. The value may either be an architected non-zero value, indicating a run-time error, such as an interface version mismatch, or 0 (zero), indicating a stub defined exception that is specified with the stub data. If a non-zero value is present, no stub data is allowed. Possible values are given in Table E-1 on page 549.

Certain status values imply that the call did not execute. To keep such status values consistent with the flag, an implementation should model all fault messages as being initialised with the PFC_DID_NOT_EXECUTE

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

flag set to TRUE, then cleared when the run-time system (or stub, if the implementation allows) passes control to the server stub routine.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4.8 The orphaned PDU

The IDL declaration of the **orphaned** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;         /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;   /* 01:01 minor version */
        u_int8  PTYPE = orphaned;     /* 02:01 orphaned PDU */
        u_int8  pfc_flags;            /* 03:01 flags */
        byte    packed_drep[4];       /* 04:04 NDR data rep format label*/
        u_int16 frag_length;          /* 08:02 total length of fragment */
        u_int16 auth_length;          /* 10:02 length of auth_value */
        u_int32  call_id;             /* 12:04 call identifier */

      /* end common fields */

      /* optional authentication verifier
         * following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier; /* xx:yy */

} rpcconn_orphaned_hdr_t;
```

The **orphaned** PDU is used by a client to notify a server that it is aborting a request in progress that has not been entirely transmitted yet, or that it is aborting a (possibly lengthy) response in progress.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

12.6.4.9 The request PDU

The IDL declaration of the **request** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;  /* 01:01 minor version */
        u_int8  PTYPE = request ;    /* 02:01 request PDU */
        u_int8  pfc_flags;           /* 03:01 flags */
        byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
        u_int16 frag_length;         /* 08:02 total length of fragment */
        u_int16 auth_length;         /* 10:02 length of auth_value */
        u_int32  call_id;            /* 12:04 call identifier */

      /* end common fields */

      /* needed on request, response, fault */

        u_int32  alloc_hint;         /* 16:04 allocation hint */
        p_context_id_t p_cont_id     /* 20:02 pres context, i.e. data rep */
        u_int16 opnum;               /* 22:02 operation #
                                      * within the interface */

      /* optional field for request, only present if the PFC_OBJECT_UUID
         * field is non-zero */

        uuid_t  object;              /* 24:16 object UID */

      /* stub data, 8-octet aligned
                    .
                    .
                    .                */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier; /* xx:yy */

} rpcconn_request_hdr_t;
```

The **request** PDU is used for an initial call request. The **p_cont_id** field holds a presentation context identifier that identifies the data representation. The **opnum** field identifies the operation being invoked within the interface.

The PDU may also contain an object UUID. In this case the PFC_OBJECT_UUID flag is set in **pfc_flags**, and the PDU includes the **object** field. If the PFC_OBJECT_UUID flag is not set, the PDU does not include the **object** field.

The **alloc_hint** field is optionally used by the client to provide a hint to the receiver of the amount of buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation. The server must work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

The minimum size of an **rpcconn_request_hdr_t** is 24 octets. If a non-nil object UUID or authentication and/or integrity or privacy services are used, the size will be larger.

The size of the stub data is calculated as follows:

```
stub_data_length = frag_length - fixed_header_length - auth_length;
if pfc_flags & PFC_OBJECT_UUID {
    stub_data_length = stub_data_length - sizeof(uuid_t);
    }
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

where the current value of **fixed_header_length** is 24 octets.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4.10 The response PDU

The IDL declaration of the **response** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
       u_int8  rpc_vers = 5;         /* 00:01 RPC version */
       u_int8  rpc_vers_minor = 0;   /* 01:01 minor version */
        u_int8  PTYPE = response;     /* 02:01 response PDU */
        u_int8  pfc_flags;            /* 03:01 flags */
        byte    packed_drep[4];       /* 04:04 NDR data rep format label*/
        u_int16 frag_length;          /* 08:02 total length of fragment */
        u_int16 auth_length;          /* 10:02 length of auth_value */
        u_int32 call_id;              /* 12:04 call identifier */

      /* end common fields */

      /* needed for request, response, fault */

        u_int32  alloc_hint;          /* 16:04 allocation hint */
        p_context_id_t p_cont_id;     /* 20:02 pres context, i.e.
                                       * data rep */

      /* needed for response or fault */

        u_int8  cancel_count          /* 22:01 cancel count */
        u_int8  reserved;             /* 23:01 reserved, m.b.z. */

      /* stub data here, 8-octet aligned
               .
               .
               .                            */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_trailer_co_t   auth_verifier; /* xx:yy */

} rpcconn_response_hdr_t;
```

The **response** PDU is used to respond to an active call. The **p_cont_id** field holds a context identifier that identifies the data representation. The **cancel_count** field holds a count of cancels received.

The **alloc_hint** field is optionally used by the transmitter to provide a hint to the receiver of the amount of buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation. The receiver must work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4.11 The shutdown PDU

The IDL declaration of the **shutdown** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;         /* 00:01 RPC version */
        u_int8  rpc_vers_minor = 0;   /* 01:01 minor version */
        u_int8  PTYPE = shutdown ;    /* 02:01 shutdown PDU */
        u_int8  pfc_flags;            /* 03:01 flags */
        byte    packed_drep[4];       /* 04:04 NDR data rep format label*/
        u_int16 frag_length;          /* 08:02 total length of fragment */
        u_int16 auth_length;          /* 10:02 */
        u_int32  call_id;             /* 12:04 call identifier */

      /* end common fields */

} rpcconn_shutdown_hdr_t;
```

The **shutdown** PDU is sent by the server to request that a client terminate the connection, freeing the related resources.

The **shutdown** PDU never contains an authentication verifier even if authentication services are in use.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Section 13. Security

This Section defines the RPC security services that an RPC application may select and describes how they are supported in the basic RPC protocol and mapped to the underlying security services. Subclause 13.1 on page 496 describes security semantics and generic security encodings that are provided by RPC for both the connection-oriented and connectionless RPC protocols. Subclause 13.2 on page 499 and Subclause 13.3 on page 505 specify encodings for the connectionless and connection-oriented RPC protocols, respectively, for the DCE secret key authentication protocol. The encodings specified in Subclause 13.2 and Subclause 13.3 are not generally applicable to other security protocols.

In addition to the encodings and semantics specified in this Section, the RPC run-time protocol machines specified in Section 9 to Section 11 indicate how security-related processing is integrated with protocol processing. The mechanisms by which the underlying security services enforce protection guarantees (for example, encryption algorithms or cryptographic key management) are outside the scope of the RPC specification. The **DCE: Security Services** specification defines these protocols, algorithms, other security-related processing, and the contents of the messages used by the underlying security services to support RPC.

The ISO C standard Security Model offers a number of optional services above the basic RPC. It offers a variety of levels of security service quality. These are realised via a combination of authentication, data protection and authorisation mechanisms.

The specified models allow DCE users to invoke cryptographically secured mutual authentication of a client and server, and to pass certified authorisation data from client to server as part of the RPC invocation. The server can then discover the client's identity and authorisation credentials, and determine what access to authorise. RPC security services also provide protection against undetected modifications of call data, cryptographic privacy of data, and protection against replay of calls and data.

The RPC run-time system has two roles in this. First, it is the conduit for exchanging the security credentials between clients and servers. Second, it may protect its communications from various security threats.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.1 The Generic RPC Security Model

The generic RPC security model and encodings specified in the following subclauses assume support for, but are not limited to, the ISO C standard Security Service. They may be applied to alternate security services as well. The model and encodings apply to both the connectionless and connection-oriented RPC protocols.

13.1.1 Generic Operation

When an RPC is initiated with a request for security services, the service provider determines if a pairwise security context has already been established between the client and server principals. This security context might include a shared session key, sequence numbers, verification state, and so on. If this context has not been established yet for that client/server principal pair, or has expired, the client RPC service provider requests initiating credentials from the security services. If these credentials are successfully acquired, the client RPC service provider then incorporates them into the initial request. The credentials may be exchanged only on call boundaries.

The server processes these security data and then responds with its own security data. If this exchange is successful, both the client and the server have established and synchronised their initial security information. Depending on the level of service requested, this may provide strong cryptographically-based mutual authentication.

For the actual call PDUs, for control PDUs and for subsequent calls, the selected protection services are provided. For example, if strong integrity protection is required, each PDU is cryptographically protected against undetected modification and deletion by the transmitter and verified by the receiver. Or, if privacy protection is specified, stub data for **request** and **response** PDUs is encrypted and integrity protected. If a mismatch is found, an error message is generated, which is encoded in a **fault**, **alter_context_response** or **bind_nak** PDU for the connection-oriented protocol or a **reject** PDU for the connectionless protocol, as detailed in Section 10 and Section 11.

Note: The level of service cannot be changed for an already established security context.

A client that represents multiple user principals has the option of providing security services such that each principal is individually authenticated. When a security context is established, a requesting client principal is authenticated. An application may obtain the security credentials for additional client principals, and then cause additional mutual authentications to occur by requesting security services on behalf of the additional principals.

For each principal, the RPC protocols exchange the credentials in one of three ways:

- by performing the conversation manager exchange for authenticated calls (connectionless protocol)
- by performing the **alter_context** exchange on the existing association (connection-oriented protocol)
- by creating a new association with the **bind** exchange (connection-oriented protocol).

The server may allow for a grace period of service after credentials expire to compensate for processing and communications delays. The client avoids sending requests it knows are likely to expire. To force a re-authentication, it fetches new credentials for use with the server from the security service and establishes the new security context according to the underlying protocol.

The RPC run-time system determines the security services used when accessing name services on behalf of a principal. Subclause 12.6.3 on page 477 describes how RPC uses name services. If the credentials of the principal are available to the RPC run-time system, then mutual authentication, based on these credentials, is used for name service accesses. If the security context of the principal is not available to the RPC run-time system, then name service accesses are unauthenticated. It is important to note that there is a difference between the security context being unavailable and the security context having expired or being otherwise invalid. An expired or invalid context results in authentication failure.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

13.1.2 Generic Encodings

Connection-oriented and connectionless PDUs may contain an optional authentication verifier. Authentication verifier encodings are largely dependent on the authentication protocol in use as detailed in the **DCE: Security Services** specification. The following subclauses specify those encodings that are authentication protocol-independent.

13.1.2.1 Protection Levels

RPC implementations are not required to support all of the protection levels specified here, but supported protection levels must have at least this level of protection for all protocols.

Authentication verifiers encode protection levels as a single byte. The defined protection levels are as follows:

dce_c_authn_level_none=1
: The client has requested that no protection be performed. Depending on server policy, the client may be granted access as an unauthenticated principal.

dce_c_authn_level_connect=2
: The client and server identities are exchanged and cryptographically verified. Strong mutual authentication is achieved — per association for the connection-oriented protocol and per activity for the connectionless protocol — and is protected against replays. However, this level provides no protection services per PDU.

dce_c_authn_level_call=3
: This level offers the **dce_c_authn_level_connect** services plus integrity protection of the first fragment only of each call. For the connection-oriented protocol any request for this level is automatically upgraded to **dce_c_authn_level_pkt**.

dce_c_authn_level_pkt=4
: This level offers the **dce_c_authn_level_connect** services plus per-PDU replay and misordering detection. It provides no per-PDU modification protection.

dce_c_authn_level_pkt_integrity=5
: This level offers the **dce_c_authn_level_pkt** services plus per-PDU modification and deletion detection.

dce_c_authn_level_pkt_privacy=6
: This level offers the **dce_c_authn_level_pkt_integrity** services plus privacy (encryption) of stub call arguments only. All run-time and lower-layer headers are still transmitted in clear text.

These values map directly to the values specified in Annex D for the **protect_level** argument to the RPC API routines. The **protect_level** value **rpc_c_protect_level_level** specifies the same protection level as the one specified by the PDU value **dce_c_authn_level_level**.

13.1.2.2 Authentication Services

Authentication services are identified by a single byte. In connectionless PDUs, this is encoded in the PDU header field **auth_proto**. In connection-oriented PDUs, this is encoded in the authentication verifier. The currently supported values are as follows:

- **dce_c_rpc_authn_protocol_none**=0
- **dce_c_rpc_authn_protocol_krb5**=1

These values map directly to the values specified in Annex D for the **authn_svc** argument to the RPC API routines. The value dce_c_rpc_authn_protocol_none maps to rpc_c_authn_none, and the value **dce_c_rpc_authn_protocol_krb5** maps to rpc_c_authn_dce_secret.

The cryptographic protocols and algorithms to which these identifiers map are defined by the **DCE: Security Services** specification.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

13.1.2.3 Authorisation Services

Authorisation services are identified by a single byte. In the connection-oriented protocol, this is encoded in the authentication verifier. In the connectionless protocol, this is part of the data of the conversation manager challenge and response.

Two authorisation models are supported. These are encoded with the following values:

- **dce_c_authz_name**=1
- **dce_c_authz_dce**=2

These values map directly to the values specified in Annex D for the **authz_svc** argument to the RPC API routines.

The authorisation service **dce_c_authz_name** asserts, without cryptographic protection, the principal name for level **dce_c_authn_level_none** and authenticates the principal name for other levels. The authorisation service **dce_c_authz_dce** asserts the principal name and authorisation data, without cryptographic protection, for level **dce_c_authn_level_none** and authenticates the principal and its authorisation data for other levels. The **DCE: Security Services** specification specifies the guarantees provided by these authorisation models.

13.1.3 Underlying Security Services Required

To support RPC security, underlying security services called by RPC must provide:

- Integrity protection of data composed of an arbitrary number of octets. The integrity protection for some RPC levels of service require that the underlying security services compute and return a value which will be referred to as a "checksum".
- Privacy protection of data composed of an arbitrary number of octets. The privacy protection for some RPC levels of service require that the underlying security services compute and return privacy protected data.
- Creation and verification of secure authentication and authorisation credentials.
- Indication as to whether an existing security context can be used for an RPC.

The methods by which these services are instantiated are defined by the **DCE: Security Services** specification. Algorithms not defined by the **DCE: Security Services** specification may be used instead to provide these services with loss of universal interoperability.

Note: In subsequent subclauses, references are made to the invocation of these services. The data passed to these services for privacy or integrity protection, or both, must have been encoded in the transfer syntax to be used in RPC PDUs. For example, when using NDR encoding, the protection operations are carried out on the sender's representation of the data.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.2 Security Services for Connection-oriented Protocol

The following subclauses specify security semantics and encodings for connection-oriented RPC protocol when the PDU header field **auth_length** is non-zero and the field **auth_type** of the authentication verifier is **dce_c_rpc_authn_protocol_krb5**. Use of other protection services is permitted but is outside the scope of this specification and will reduce interoperability.

13.2.1 Client Association State Machine

Whenever a client attempts to create a new association to a server, it must take the following steps:

a) Create a new association UUID, **assoc_uuid**.

b) Invoke the security services to compute a non-cryptographic checksum of the **assoc_uuid**. The value computed will be denoted as **assoc_uuid_chksum**. The algorithm for computing this non-cryptographic checksum is specified in the **DCE: Security Services** specification.

c) Initialise two 32-bit state variables called **sequence numbers** which may assume only integral values in the range 0 to $2^{32}-1$ inclusive:

- **u_int32 next_send_seq**=0
- **u_int32 next_recv_seq**=0

These steps are required even if there are already other associations established and authenticated for the client/server pair. Each new association request **must** establish pairwise credentials between the client and server even if they have already been established for the same client/server pair on a different association.

While establishing security credentials, the client provider may transmit a **bind** PDU to establish the initial security context, or an **alter_context** PDU to alter or add new security contexts. It may receive **bind_ack, bind_nak** or **alter_context_resp** PDUs.

13.2.2 Server Association State Machine

Whenever a server receives a new association request, it must take the following steps:

a) Store the received value **assoc_uuid_chksum**.

b) Initialise two 32-bit state variables called **sequence numbers** which may assume only integral values in the range 0 to $2^{32}-1$ inclusive:

- **u_int32 next_send_seq**=0
- **u_int32 next_recv_seq**=1

While establishing security credentials, the server provider may receive **bind** or **alter_context** PDUs, and transmit **bind_ack**, **bind_nak** or **alter_context_resp** PDUs.

13.2.3 Sequence Numbers

While sequence numbers are not transmitted explicitly by the RPC protocol, they are used in the computation of various security checks for the PDUs. Sequence numbers are initialised to 0 (zero) on the establishment of each association and used by all PDUs even when no security services have been requested. Every PDU transmitted, including call fragment PDUs, is assigned the value of **next_send_seq,** which is then atomically incremented. Implementations must ensure that PDUs are transmitted in the same order that the sequence numbers are assigned. Receivers must check the anticipated **next_recv_seq** value against the received PDUs, and then atomically increment **next_recv_seq**. Any out-of-order PDUs generate an invalid_checksum reject status code, which is transmitted in the **fault**, **bind_ack** and **alter_context_response** PDUs. Implementations must take care to verify the received PDU sequence numbers in the same order the PDUs are received.

The most significant bit of the sequence number is used as a direction indicator. When requesting security services using the sequence numbers assigned to PDUs sent from the server to the client, RPC must invert the most significant bit of the sequence number before passing the value to the underlying security services. Both client and server RPC run-time systems must perform this inversion for security processing of PDUs

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

sent from server to client.

There is no provision for overflow of sequence numbers. The maximum value is $2^{32}-1$.

13.2.4 The auth_context_id Field

The **auth_context_id** field is an unprotected hint that is transmitted to suggest the appropriate security context for the receiver to use. It may be used to distinguish among multiple user principals sharing the same client. This is typically a table index or pointer value that must be unique at least across the scope and lifetime of an association group.

13.2.5 Integrity Protection

The integrity checksums required for some of the levels of service may be computed via different algorithms. The algorithm used to protect a specific PDU is indicated by the value of the **sub_type** field, which encodes the authentication service using one of the values given in Subclause 13.1.2.2 on page 497, and the desired protection level.

For all variants of the checksum, the transmitting side must pass to the underlying security services:

- the security context indicated by the current **auth_context_id**
- the **assoc_uuid_chksum**
- the desired protection level
- the identification of the checksum algorithm to be used (corresponding to the value to be sent in the **sub_type** field)
- the sequence number (**next_send_seq**)
- the PDU without the authentication verifier.

The computed checksum and checksum length are then inserted into their respective authentication verifier fields.

The receiving side recomputes the checksum, by invoking the underlying security services and passing (as needed):

- the security context for the call (hinted at by the **auth_context_id** in the authentication verifier)
- the previously received **assoc_uuid_chksum**
- the desired protection level
- the identification of the checksum algorithm to be used (corresponding to the value to be received in the **sub_type** field)
- the local receive sequence number, **next_recv_seq**
- the received PDU without the authentication verifier.

The receiver then compares the computed checksum to the value received in the verifier. If they are identical, then the PDU is accepted as authentic.

If the receiver does not support the **sub_type** specified by the transmitter, an error indicating invalid checksum is returned. The callee should respond with the same subtype requested by the caller.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

13.2.6 Connection-oriented Encodings

Most of the connection-oriented RPC PDUs, as defined in Section 12, may include an optional authentication verifier that contains authentication and/or authorisation data. The verifier is present if and only if the **auth_length** field in the PDU is non-zero. The verifier consists of a set of common fields and one field, **auth_value**, the encoding of which depends on authentication service, authorisation service, protection level and PDU type. The length of the authentication verifier varies depending on the data encoded by the **auth_value** field.

13.2.6.1 Common Authentication Verifier Encodings

The common authentication verifier is defined as the following structure:

```
typedef struct{

        /* restore 4 byte alignment */

        u_int8  [size_is(auth_pad_length)] auth_pad[]; /* align(4) */
        u_int8  auth_type;       /* :01  which authent service */
        u_int8  auth_level;      /* :01  which level within service */
        u_int8  auth_pad_length; /* :01 */
        u_int8  auth_reserved;   /* :01 reserved, m.b.z. */
        u_int32 auth_context_id; /* :04 */
        u_int8  [size_is(auth_length)] auth_value[]; /* credentials */

} auth_verifier_co_t;
```

The **auth_pad** field is required to restore 0 mod 4 alignment following the stub data, if any. It consists of 0, 1, 2 or 3 null bytes.

The **auth_type** field defines which authentication service is in use. Currently supported values are specified in Subclause 13.1.2.2 on page 497.

The **auth_level** field defines the protection level. The supported values are specified in Subclause 13.1.2.1 on page 497.

The **auth_pad_length** field indicates the number of pad bytes that are appended to the header and stub data before the authentication verifier.

The **auth_reserved** field is reserved for future use. It must be 0 (zero) on transmission, and it is ignored on reception.

The **auth_context_id** field indicates the corresponding security context previously established.

The **auth_value** field may contain a variety of security-related data. For the **bind**, **bind_ack**, **alter_context** and **alter_context_response** PDUs, this field encodes credentials. For other PDUs, this field holds checksums and other per-PDU security data that depend on the protection level. Encodings of this field depend on the PDU type, authentication service, authorisation service and protection level. Subclause 13.2.6.2 specifies the encodings of this field for per-PDU security services. Subclause 13.2.6.3 on page 503 specifies the connection-oriented encodings for exchanging credentials.

13.2.6.2 Encoding for Per-PDU Services

Authentication verifiers contain an **auth_value** field that holds checksums, credentials and other security-related data. When used for per-PDU security services, the **auth_value** encoding depends on the protection level. The following subclauses define the encodings of the **auth_value** field to provide per-PDU security services. For protection level **dce_c_authn_level_pkt_privacy**, the encryption of the PDU body data is also specified.

The encodings are modelled as IDL structure definitions. As in the RPC PDU definitions, they assume no padding between elements, and they assume NDR transfer syntax.

The **sub_type** fields in **auth_value** encodings allow variant algorithms for providing the same level of services. This field can also be used to indicate that an invalid checksum was received. The currently defined values are as follows:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- **dce_c_cn_dce_sub_type**=0
- **dce_c_cn_dce_sub_type_md5**=1
- **dce_c_cn_dce_sub_type_invalid_checksum**=2 (Invalid integrity)

The mapping from the values **dce_c_cn_dce_sub_type** and **dce_c_cn_dce_sub_type_md5** to specific algorithms is defined by the **DCE: Security Services** specification. The value **dce_c_cn_dce_sub_type_invalid_checksum** is used to indicate that an invalid checksum was detected.

The following subclauses specify **auth_value** encodings for each protection level.

The dce_c_authn_level_none Protection Level

The **auth_value** is null; the entire authentication verifier is omitted.

The dce_c_authn_level_connect Protection Level

The **auth_value** encoding is as follows:

```
typedef struct{
            u_int8      sub_type;
            u_int8      reserved;       /* must be zero. */
            } auth_value_t;
```

The dce_c_authn_level_call Protection Level

This level is not supported as a separate entity. Instead, requests for this level will automatically be upgraded to **dce_c_authn_level_pkt**.

The dce_c_authn_level_pkt Protection Level

The **auth_value** encoding is as follows:

```
typedef struct{
            u_int8      sub_type;
            u_int8      checksum_length;
            byte        [size_is(checksum_length)] checksum[];
            } auth_value_t;
```

where the field "checksum" is the checksum value returned by the underlying security service in response to an integrity protection call (see Subclause 13.1.1 on page 496).

The dce_c_authn_level_pkt_integrity Protection Level

The **auth_value** encoding is as follows:

```
typedef struct{
            u_int8      sub_type;
            u_int8      checksum_length;
            byte        [size_is(checksum_length)] checksum[];
            } auth_value_t;
```

where the field "checksum" is the checksum value returned by the underlying security service in response to an integrity protection call (see Subclause 13.1.1 on page 496).

The dce_c_authn_level_pkt_privacy Protection Level

In contrast to the other security levels, this level also requires changes to the contents of the body data of the standard RPC PDUs. This level of service provides strong integrity protection for the entire PDU, plus privacy protection for the body data only. Therefore, only the bodies of the **request**, **response** and **fault** PDUs are encrypted.

The **auth_value** encoding is as follows:

```
typedef struct{
            u_int8      sub_type;
            u_int8      checksum_length;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
byte        [size_is(checksum_length)] checksum[];
} auth_value_t;
```

The PDU to be protected is divided into two pieces: the RPC header and the PDU body composed of data generated by the stub's marshalling procedures. The following is passed to underlying security service in a call requesting privacy protection:

- the RPC header
- the PDU body, if any
- the desired checksum algorithm (corresponding to the value to be sent in the **sub_type** field).

The underlying security service returns the privacy protected PDU body, if any, and a checksum value.

If there was a PDU body, then the privacy protected PDU body replaces the original, unprotected PDU body in the PDU. The returned checksum value is inserted into the authentication verifier checksum field. The resulting PDU may then be transmitted.

If there was no PDU body, then the checksum value is inserted into the checksum field of the authentication verifier. The resulting PDU may then be transmitted.

The **DCE: Security Services** specification defines the algorithms used to create the protected PDU body and the checksum value.

13.2.6.3 Credentials Encoding

This subclause defines the contents of the optional **auth_value** fields in the **bind, bind_ack, alter_context** and **alter_context_response** PDUs as used for establishing credentials. The **auth_value** fields are modelled as IDL structure definitions. As in the RPC PDU definitions, the definitions assume no padding between elements, and they assume NDR transfer syntax.

For any protection level requested, including **dce_c_authn_level_none**, a **bind** PDU includes optional authentication data. If a new client principal is being introduced, either a **bind** for a new association or an **alter_context** on an existing association is used. The mutual authentication response is carried by the **bind_ack** or **alter_context_response**, respectively.

When security services are in effect, the **credentials** field is empty (**cred_length**=0) for an **rpc_alter_context** or **rpc_alter_context_response** PDU that is intended only to change non-security-related context, such as presentation context, transfer syntax, and the like.

The generic encoding is as follows:

```
typedef struct{
            u_int32     assoc_uuid_crc; /* checksum of assoc_uuid */
            u_int8      sub_type;
            u_int8      checksum_length;
            u_int16     cred_length;
            byte [size_is(cred_length)] credentials[];
            byte [size_is(checksum_length)] checksum[];
            } auth_value_t;
```

where:

- The **assoc_uuid_crc** field is defined by the client to be the value **assoc_uuid_chksum**, and is ignored on response.
- The **credentials** field depends on the **auth_type**, as specified in **DCE Secret Key credentials Field Encoding** on page 504.
- The **checksum** field depends on the level of service, as specified in **DCE Secret Key credentials Field Encoding** on page 504.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DCE Secret Key credentials Field Encoding

The following algorithm defines the **credentials** field encoding for **auth_type=dce_c_rpc_authn_protocol_krb5**:

For a **bind** or **alter_context** PDU with authentication level **dce_c_authn_level_none**, authentication service of **dce_c_rpc_authn_protocol_krb5**.

- If the authorisation service is **dce_c_authz_name**, the **credentials** field has the form:

```
u_int8  authz_type=dce_c_authz_name;
char    name[];      /* null-terminated local principal name */
                     /* size auth_length-1 */
```

- If the authorisation service is **dce_c_authz_dce**, the **credentials** field has the form:

```
u_int8   authz_type=dce_c_authz_dce;
byte     pac[];
```

  where the contents of the **pac** field is determined by the underlying security services and defined in the **DCE: Security Services** specification.

For a **bind_ack** or **alter_context_response** PDU with authentication level **dce_c_authn_level_none**, authentication service of **dce_c_rpc_authn_protocol_krb5**, and authorisation service of either **dce_c_authz_name** or **dce_c_authz_dce**, the **credentials** field has one of the following forms:

- If no error has occurred, then the **auth_value** field of the authentication verifier is empty (null).
- If an authentication error occurred then the **auth_value** field contains:

```
u_int32 statusq  /* big-endian encoded */
```

For a **bind** or **alter_context** PDU with any authentication level **except dce_c_authn_level_none**, authentication service of **dce_c_rpc_authn_protocol_krb5**, and authorisation service of either **dce_c_authz_name** or **dce_c_authz_dce**, then the **credentials** field has the form:

```
byte   request[];
```

where the contents of the request field is determined by the underlying security services and defined in the **DCE: Security Services** specification.

For a **bind_ack** or **alter_context_response** PDU with any authentication level **except dce_c_authn_level_none**, authentication service of **dce_c_rpc_authn_protocol_krb5**, and authorisation service of either **dce_c_authz_name** or **dce_c_authz_dce**, then the **credentials** field has one of the following forms:

- If no error has occurred, the **credentials** field is encoded as

```
byte   response[];
```

  where the contents of the **response** field are determined by the underlying security services and defined in the **DCE: Security Services** specification.

- If an authentication error occurred, the **credentials** field is encoded as

```
byte   error[];
```

  where the contents of the **error** field are determined by the underlying security services and defined in the **DCE: Security Services** specification.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.3 Security Services for Connectionless Protocol

The following subclauses specify security semantics and encodings for connectionless RPC protocol when the PDU header field **auth_proto=dce_c_rpc_authn_protocol_krb5**. Use of other protection services is permitted but outside the scope of this specification.

13.3.1 Server Receive Processing

On receiving a PDU, the connectionless protocol machine first locates the activity record that is associated with the client. This is determined through the activity ID. If no activity record is found, one is created, and if a security service is requested, the challenge/response exchange is initiated by performing the conversation manager callback for authenticated calls.

The protocol machine verifies the conversation manager callback **response** PDU. If the appropriate values of the this PDU match the authentication information of the activity record (refer to the **DCE: Security Services** specification), the security context is established; otherwise, an error PDU (**reject** PDU) is generated.

The server also initiates the challenge/response exchange if it cannot locate the received session key that is associated with the key sequence number.

Based on a valid security context, the server verifies the following for each received PDU:

- the authentication verifier according to the specific authentication protocol
- the selected level of per-PDU service.

If mismatches are detected, error PDUs (**fault** PDUs) are generated.

If the fragment number in the **request** PDU is 0 (zero), indicating that the PDU is the first PDU of a call, the current time is compared with the expiration time of the security context. If it has expired, an error PDU is generated.

13.3.2 Client Receive Processing

Client receive processing is identical to server receive processing, except that no attempt is made to learn the key through a challenge/response exchange.

13.3.3 Conversation Manager Encodings

The **conv_who_are_you_auth**( ) operation of the conversation manager provides a way to piggyback a variable-length array of bytes on each leg of the operation by using a challenge/response exchange. The formats of these conversation manager challenge **request** and **response** PDUs when **auth_proto=dce_c_rpc_authn_protocol_krb5** are specified in the following subclauses.

13.3.3.1 Challenge Request Data Encoding

The conversation manager challenge **request** PDU, which is generated by the server, is entirely in plaintext because the server does not necessarily share any keys with the client. It is transferred in the **in_data** parameter (that is, as the stub data of an RPC **request** PDU) of the **conv_who_are_you_auth**( ) operation. When **auth_proto=dce_c_rpc_authn_protocol_krb5**, it is 12 bytes long and consists of:

```
typedef struct {
      u_int32             key_seq_num; /* big endian */
      byte[8]             challenge;
      } in_data;
```

key_seq_num The sequence number of the key requested by the server.

challenge A 64-bit random value. See the **DCE: Security Services** specification for information on generating this **confounder**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

13.3.3.2 Response Data Encoding

The response is transferred as the **out_data** parameter (that is, as stub data of an RPC **response** PDU) of the **conv_who_are_you_auth**( ) conversation manager operation. The contents of this PDU are specified in the **DCE: Security Services** specification.

13.3.4 Authentication Verifier Encodings

Connectionless PDUs contain an authentication verifier if the PDU header field **auth_proto** is non-zero. Otherwise, the authentication verifier is not present.

The encoding and length of the authentication verifier depends on the authentication service, as identified by the PDU header field **auth_proto**.

The authentication data encodings for the PDU authentication verifier are specified in the following subclauses for **auth_proto=dce_c_rpc_authn_protocol_krb5**.

Unless specified otherwise, the data types and values are encoded in the NDR transfer syntax. Note that no padding between elements within a data structure is assumed and that the alignment requirements for the PDU header (see Subclause 12.3 on page 466) also apply to the authentication verifier of the PDU trailer.

When the PDU header field **auth_proto=dce_c_rpc_authn_protocol_krb5**, every PDU contains a 20 or 24-byte authentication verifier. The first three fields of the verifier consist of a plaintext header followed by an 16-byte ciphertext authentication value, as follows:

```
typedef struct {
      u_int8              protection_level;
      u_int8              key_vers_num;
      byte[pad_length]    pad;
      byte[16]            auth_value;
      } auth_trailer_cl_t;
```

protection_level The protection level of the RPC. It indicates the level of service as determined by the protection level values (see Subclause 13.1 on page 496).

key_vers_num The version number of the key that indicates the key used to encrypt or to calculate the checksum of any ciphertext in the authentication value.

pad A padding field whose value is all zeros. The length of this array (**pad_length**) is 6 bytes for protection level **dce_c_authn_level_privacy**, 2 bytes otherwise.

auth_value The ciphertext of the authentication verifier. The format of **auth_value** depends on the level of service. The plaintext is encoded in the transfer syntax as specified in the PDU header field **drep**.

The following subclauses describe the authentication value encodings for each protection level.

13.3.4.1 dce_c_authn_level_none

There is no authentication verifier in the PDU for this protection level.

13.3.4.2 dce_c_authn_level_connect

The **auth_value** field of the verifier is ignored for this level of service, which does not provide protection per PDU.

13.3.4.3 dce_c_authn_level_call

This level is not supported. Requests for this level will automatically be upgraded to **dce_c_authn_level_pkt**.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

13.3.4.4 dce_c_authn_level_pkt

For per-PDU level, the underlying security service computes a 8-octet checksum of a plaintext that is supplied to the **auth_value** field of the authentication verifier. The plaintext is constructed as follows:

```
typedef struct {
        u_int32  seqnum;
        u_int32  fragnum;
        } plaintext;
```

seqnum The sequence number of the call, as specified in the PDU header. If the server generates the authentication verifier, the high-order bit of the sequence number is set to 1, as indication for the direction.

fragnum The fragment number of the call, as specified in the PDU header.

The **DCE: Security Services** specification defines the algorithms used to create the checksum value for the **auth_value** field.

13.3.4.5 dce_c_authn_level_integrity

For PDU-integrity level, the underlying security service computes a 16-octet checksum of the concatenated PDU header and body data that is supplied to the **auth_value** field of the authentication verifier.

The **DCE: Security Services** specification defines the algorithms used to create the checksum value for the **auth_value** field.

13.3.4.6 dce_c_authn_level_privacy

In contrast to the other security levels, this level also requires changes to the contents of the body data of the standard RPC PDUs. This level of service provides strong integrity protection for the entire PDU, plus privacy protection for the body data only.

The PDU to be protected is divided into two pieces: the RPC header and the PDU body composed of data generated by the stub's marshalling procedures. The following is passed to the underlying security service in a call requesting privacy protection:

- the RPC header
- the PDU body, if any
- the checksum field that was supplied as part of the **out_data** parameter in the conversation manager operation
- the sequence number of the call, as specified in the PDU header.

The underlying security service returns the privacy protected PDU body, if any, and a 16-octet checksum value.

If there was a PDU body, then the privacy protected PDU body replaces the original, unprotected PDU body in the PDU. Insert the returned checksum value into the **auth_value** field. The resulting PDU may then be transmitted.

If there was no PDU body, then insert the returned checksum value into the **auth_value** field. The resulting PDU may then be transmitted.

The **DCE: Security Services** specification defines the algorithms used to create the protected PDU body and the checksum value.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

This page intentionally left blank.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Section 14. Transfer Syntax NDR

Most application programs treat procedure call inputs and outputs as values of structured data types such as integers, arrays and pointers. One role of IDL is to provide syntax for describing these structured data types and values. However, the RPC protocol specifies that inputs and outputs be passed in octet streams. The role of NDR is to provide a mapping of IDL data types onto octet streams. NDR defines primitive data types, constructed data types and representations for these types in an octet stream.

For some primitive data types, NDR defines several data representations. For example, NDR defines ASCII and EBCDIC formats for characters. When a client or server sends an RPC PDU, the formats used are identified in the format label of the PDU. The data representation formats and the format label support the NDR **multi-canonical** approach to data conversion; that is, there is a fixed set of alternate representations for data types.

This Section describes:

- the NDR format label
- the set of NDR primitive data types and the supported data representation formats for these types
- the set of NDR constructed data types and their representations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.1 Data Representation Format Label

The NDR format label is a vector of 4 octets that identifies the particular data representation formats used to represent primitive values both in the header and in the body of an RPC PDU. The format label is itself part of the PDU header. (See Section 12 for definitions of RPC PDUs.)

Figure 14-1 illustrates the NDR format label. The four most significant bits of octet 0 indicate integer format and endian type of the floating-point representation. The four least significant bits of octet 0 indicate character format. Octet 1 indicates floating-point representation format. Octets 2 and 3 are reserved for future use and must be zero octets.

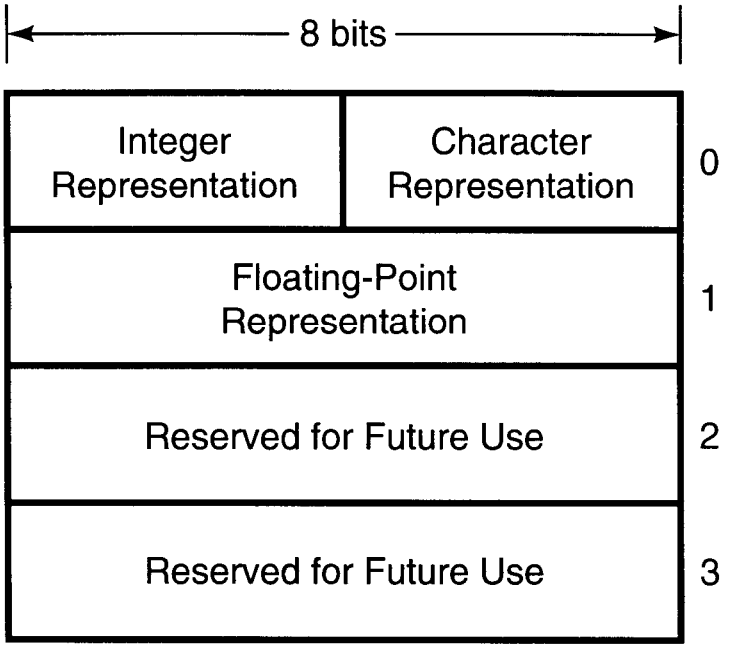


Figure 14-1 NDR Format Label

Table 14-1 lists the values associated with integer, character and floating-point formats. The values are represented in the format label in unsigned integer binary format.

Table 14-1 NDR Format Label Values

Data Type	Value in Label	Format
Character	0	ASCII
	1	EBCDIC
Integer and floating-point byte order	0	Big-endian
	1	Little-endian
Floating-point representation	0	IEEE
	1	VAX
	2	Cray
	3	IBM

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.2 NDR Primitive Types

NDR defines a set of 13 primitive data types to represent Boolean values, characters, four sizes of signed and unsigned integers, two sizes of floating-point numbers, and uninterpreted octets.

For characters, integers and floating-point numbers, NDR defines more than one representation format. The formats used in an RPC PDU are identified in the NDR format label.

All NDR primitive data types are multiples of octets in length. A octet is 8 bits. A bit can take the values 0 and 1.

14.2.1 Representation Conventions

The figures representing NDR primitive types adopt the following conventions:

- Each octet is represented by a rectangular box.
- When the figure refers to individual bits or groups of bits within an octet, the octet box is divided by vertical lines into one or more smaller rectangles that represent the individual bits or groups of bits.
- Within octets, bits and groups of bits are represented with the most significant bit at the left and the least significant bit at the right.
- Most significant bit is abbreviated MSB, and least significant bit is abbreviated LSB.
- Data types larger than one octet are depicted as a series of octet boxes arranged vertically to form a larger rectangle. The octets are ordered from top to bottom: the topmost octet appears first in the octet stream and the bottommost octet appears last.
- Bit and octet numbering, for reference purposes, begins with 0.
- Diagrams do not depict the specified alignment gaps, which can appear in the octet stream before an item (see Subclause 14.2.2.)
- When a bit is **set**, it has the value 1. When a bit is **reset**, it has the value 0.

Following the preceding rules, the order of octets, as they occur in an octet stream, can be read by reading vertically from the top octet box down to the bottom octet box. The order of bits and groups of bits in an octet, from most significant to least significant, can be read beginning at the leftmost end of an octet box and reading across to the right end. The order of bits and groups of bits in the octets of a data type can therefore be read by reading the bits from left to right in each octet, beginning with the top octet and ending with the bottom octet.

Note: Although NDR specifies the order of bits and groups of bits within the octet stream of some data types, it specifies an octet stream representation of data rather than a bit stream representation of data. NDR does not specify how a given octet stream is represented as a bit stream, which is typically the province of underlying network layers.

14.2.2 Alignment of Primitive Types

NDR enforces **NDR alignment** of primitive data; that is, any primitive of size **n** octets is aligned at a octet stream index that is a multiple of **n**. (In this version of NDR, **n** is one of {1, 2, 4, 8}.) An octet stream index indicates the number of an octet in an octet stream when octets are numbered, beginning with 0, from the first octet in the stream. Where necessary, an alignment gap, consisting of octets of unspecified value, precedes the representation of a primitive. The gap is of the smallest size sufficient to align the primitive.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.2.3 Booleans

A Boolean is a logical quantity that assumes one of two values: TRUE or FALSE. NDR represents a Boolean as one octet. It represents a value of FALSE as a **zero octet**, an octet in which every bit is reset. It represents a value of TRUE as a **non-zero octet**, an octet in which one or more bits are set.

Figure 14-2 illustrates the boolean data type as it appears in the octet stream.

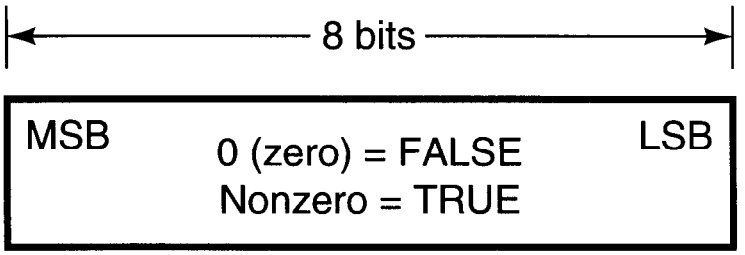


Figure 14-2 The Boolean Data Type

14.2.4 Characters

NDR represents a character as one octet. Characters have two representation formats: ASCII and EBCDIC.

Figure 14-3 illustrates the character type as it appears in the octet stream.

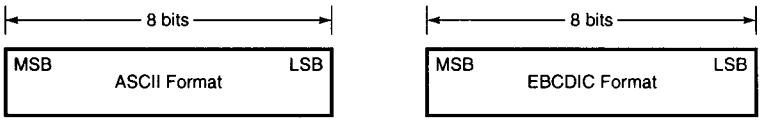


Figure 14-3 Character Data Type

14.2.5 Integers and Enumerated Types

NDR defines both signed and unsigned integers in four sizes:

small An 8-bit integer, represented in the octet stream as 1 octet.

short A 16-bit integer, represented in the octet stream as 2 octets.

long A 32-bit integer, represented in the octet stream as 4 octets.

hyper A 64-bit integer, represented in the octet stream as 8 octets.

NDR represents signed integers in twos complement format and represents unsigned integers as unsigned binary numbers. There are two integer formats: big-endian and little-endian. If the integer format is big-endian, the octets of the representation are ordered in the octet stream from the most significant octet to the least significant octet. If the integer format is little-endian, the octets of the representation are ordered in the octet stream from the least significant octet to the most significant octet.

Figure 14-4 on page 513 illustrates the integer types in big-endian and little-endian format.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

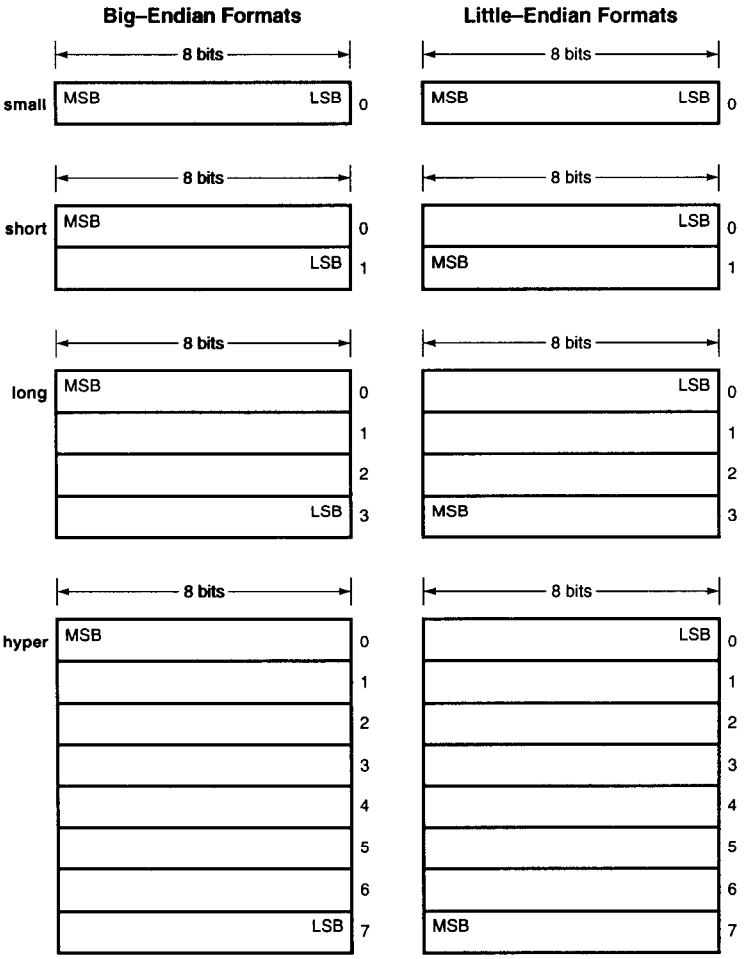


Figure 14-4 NDR Integer Formats

14.2.5.1 Enumerated Types

NDR represents enumerated types as signed short integers (2 octets).

14.2.6 Floating-point Numbers

NDR defines single-precision and double-precision floating-point data types. It represents single-precision types in 4 octets and double precision types in 8 octets.

NDR supports the following floating-point data representation formats for single-precision and double-precision floating-point types:

- IEEE single-precision and double-precision floating-point formats, which comply with the IEEE 754 standard.
- VAX **F_floating** and **G_floating** formats as defined in the VAX11 Architecture document.
- Cray floating-point format, as defined in the International Standard.tion produced by Cray Research, Inc.
- IBM short and long formats, as defined in the System/370 document.

Table 14-2 on page 514 is a conversion chart that shows how NDR single-precision and double-precision floating-point types correspond to the supported floating-point formats.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 14-2 NDR Floating Point Types

NDR Values	Conversion Values			
	IEEE	VAX	Cray	IBM
Single	Single (4 octets)	F (4 octets)	Single (4 octets)	Short (4 octets)
Double	Double (8 octets)	G (8 octets)	Double (8 octets)	Long (8 octets)

The representation of a floating-point number comprises three fields:

- The sign bit, which indicates the sign of the number. Values 0 and 1 represent positive and negative, respectively. This field is 1 bit in length.
- The exponent of the number (base 16 in IBM format, base 2 in all others), biassed by an excess. The size of this field varies according to the format, as does the excess.
- The fractional part of the number's mantissa (base 16 in IBM format, base 2 in all others). This field is also called the number's coefficient. The size of this field varies according to the format.

NDR allows implementations to use different degrees of precision in representing floating-point numbers. When the receiver is unmarshaling a floating-point number, and the number cannot be represented exactly in the receiver's floating-point format, the received (input) data are rounded such that the representable value nearest the infinitely precise result is delivered. If two representable values are equally near, the one with its least significant bit 0 (zero) is delivered.

The integer representation field of the NDR format label indicates whether floating-point values are transmitted in big-endian or little-endian format. See Table 14-1 on page 510 for the mapping between format label values and data representations.

The remainder of this subclause describes how floating-point numbers are represented in the octet stream in IEEE, VAX, Cray and IBM formats.

14.2.6.1 IEEE Format

Single-precision IEEE floating-point format is 32 bits in length, consisting of a 1-bit sign, an 8-bit exponent field (excess 127), and a 23-bit mantissa that represents a fraction in the range $1.0 \leq \mathbf{m} < 2.0$. Double precision IEEE floating-point format is 64 bits in length with a 1-bit sign, an 11-bit exponent (excess 1023), and a 52-bit mantissa.

IEEE floating-point numbers are used on machines made by a variety of manufacturers and based on a variety of architectures. Some of these machines use little-endian representation for integers; other machines use big-endian representation. When a recipient interprets an NDR octet stream whose format label specifies IEEE floating-point format, it uses the integer representation in the format label to determine the octet order of the IEEE floating-point number.

Figure 14-5 illustrates IEEE single-precision floating-point format in big-endian and little-endian integer representation. The exponents and fractions shown in this figure are represented in unsigned-binary format.

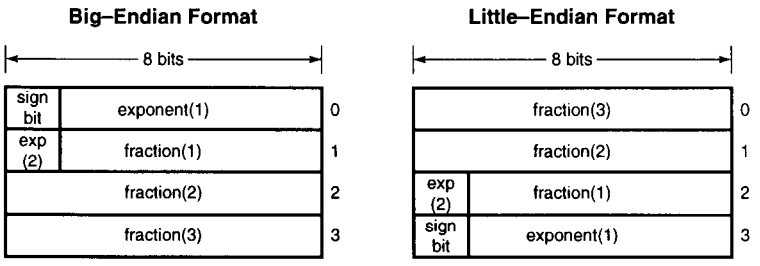


Figure 14-5 IEEE Single-precision Floating-point Format

Figure 14-6 on page 515 illustrates IEEE double-precision floating-point format in big-endian and little-endian integer representation. The exponents and fractions shown in this figure are represented in unsigned-binary format.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

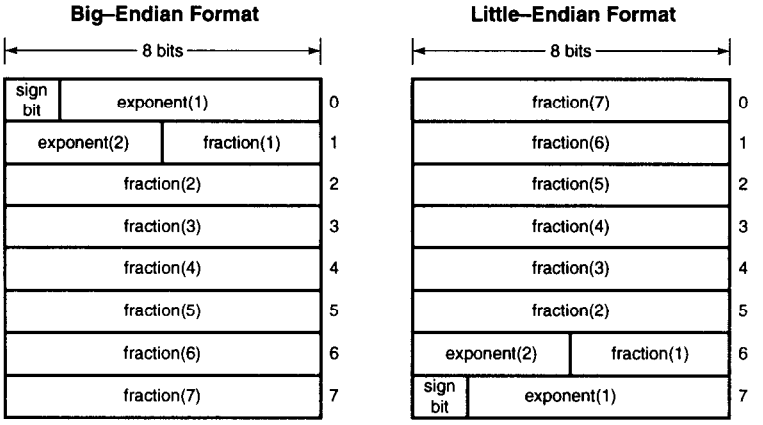


Figure 14-6 IEEE Double-precision Floating-point Format

14.2.6.2 VAX Format

VAX architecture defines four floating-point formats: **F_floating**, **D_floating**, **G_floating** and **H_floating**. **F_floating** format is 32 bits in length, including a 1-bit sign, an 8-bit exponent (excess 128), and a 23-bit mantissa that represents a fraction in the range $0.5 \leq \mathbf{m} < 1.00$. **D_floating** format is 64 bits, with a 1-bit sign, an 8-bit exponent and a 56-bit mantissa. **G_floating** format is also 64 bits, with a 1-bit sign, an 11-bit exponent (excess 1024) and a 52-bit mantissa. **H_floating** format is 128 bits.

Although the VAX architecture supports four floating-point formats, NDR uses only VAX **F_floating** format to represent VAX single-precision floating-point numbers and VAX **G_floating** format to represent VAX double-precision floating-point numbers.

Figure 14-7 illustrates VAX F floating-point representation as it appears in the octet stream. Exponents and fractions shown in this and the next figure are represented in unsigned-binary format.

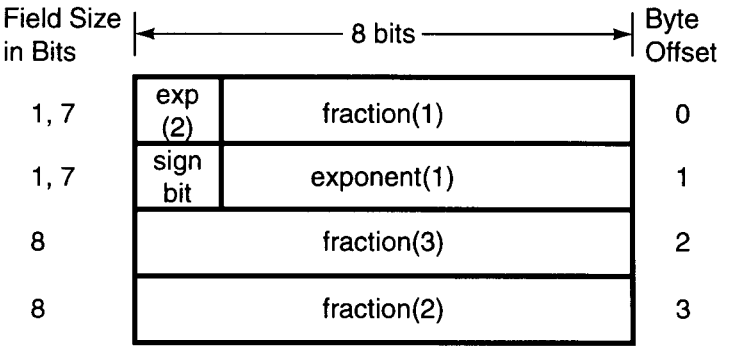


Figure 14-7 VAX Single-precision (F) Floating-point Format

Figure 14-8 on page 516 illustrates VAX G floating-point representation as it appears in the octet stream.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

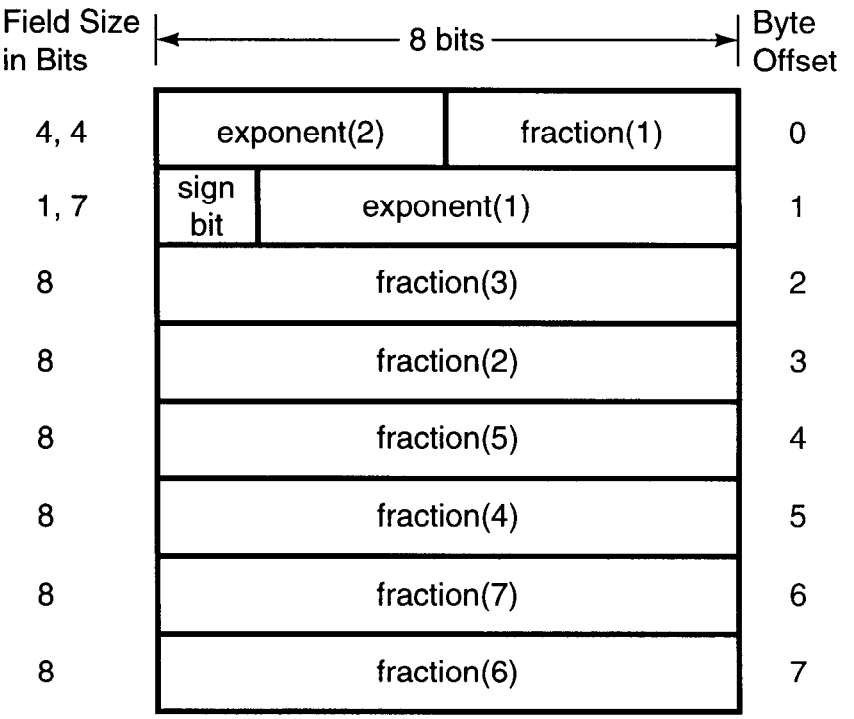


Figure 14-8 VAX Double-precision (G) Floating-point Format

Figure 14-7 on page 515 and Figure 14-8 illustrate VAX **F_floating** and **G_floating** formats using a little-endian representation for integers. However, some machines may implement VAX floating-point format with a big-endian representation. When a recipient interprets an NDR octet stream whose format label specifies VAX floating-point format, it uses the integer representation in the format label to determine the octet order of the floating-point number.

14.2.6.3 Cray Format

Cray machine architecture defines only a double-precision floating-point format. However, because Cray machines may be required to handle single-precision floating-point values (for instance, if single-precision values are specified in an interface definition), NDR defines a single-precision floating-point format for Cray machines; this format is identical to IEEE big-endian single-precision format.

A Cray double-precision floating-point number is 64 bits in length and consists of a 1-bit sign, a 15-bit exponent (16,384 excess) and a 48-bit fraction. A Cray single-precision floating-point number is 32 bits in length and consists of a 1-bit sign, an 8-bit exponent (excess 127) and a 23-bit mantissa that represents a fraction in the range $1.0 \leq \mathbf{m} < 2.0$.

Figure 14-9 on page 517 illustrates the Cray floating-point formats.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

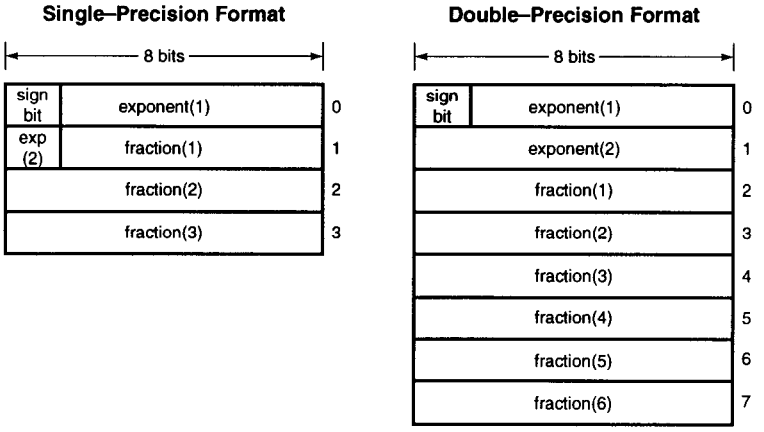


Figure 14-9 Cray Floating-point Formats

14.2.6.4 IBM Format

The IBM architecture defines short and long floating-point formats for single-precision and double-precision floating-point values, respectively. An IBM short floating-point number consists of a 1-bit sign, a 7-bit exponent and a 24-bit fraction. An IBM long floating-point number consists of a 1-bit sign, a 7-bit exponent and a 56-bit fraction. The IBM formats represent both the exponent and the fraction in hexadecimal rather than binary notation. Consequently, the exponent is base 16, while the fraction is composed of either six 4-bit hexadecimal digits or fourteen 4-bit hexadecimal digits.

Figure 14-10 illustrates the IBM short and long floating-point formats.

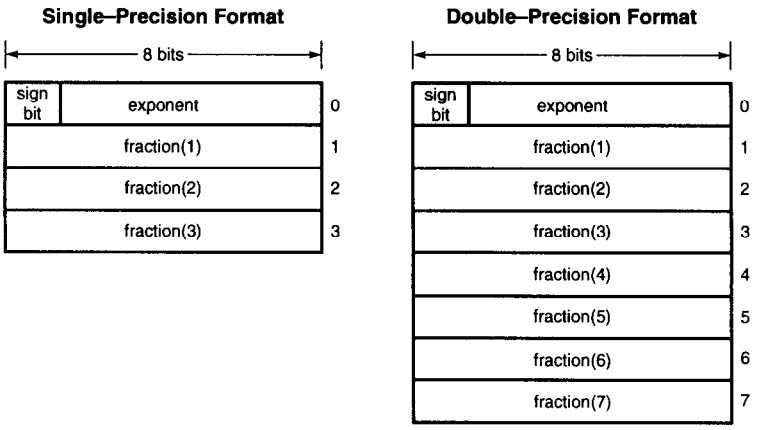


Figure 14-10 IBM Floating-point Formats

Figure 14-10 illustrates IBM floating-point format using a big-endian representation for integers. However, some machines may implement IBM floating-point with a little-endian representation. When a recipient interprets an NDR octet stream whose format label specifies IBM floating-point format, it uses the integer representation in the format label to determine the octet order of the floating-point number.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.2.7 Uninterpreted Octets

NDR defines an uninterpreted octet data type for which no internal format is defined and on which no format conversions are made.

illustrates the uninterpreted octet type as it appears in the octet stream.

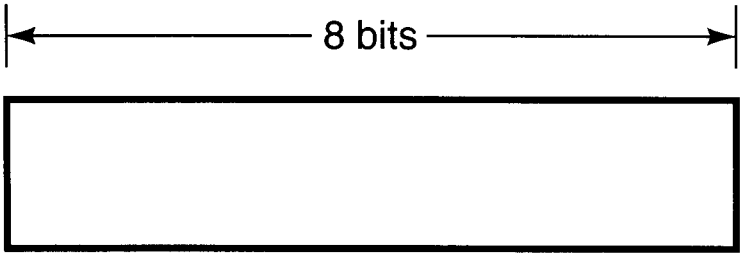


Figure 14-11 Uninterpreted Octet Representation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3 NDR Constructed Types

NDR supports data types that are constructed from the NDR primitive data types described in the previous subclause. The NDR constructed types include arrays, strings, structures, unions, variant structures, pipes and pointers.

NDR represents every NDR constructed type as a sequence of NDR primitive values. The representation formats for these primitive values are identified in the NDR format label.

All NDR constructed data types are integer multiples of octets in length.

14.3.1 Representation Conventions

The figures representing NDR constructed types adopt the following conventions:

- Constructed types are represented as a series of one or more primitive types. Each primitive type is shown as a rectangular box.
- The octet stream is depicted from left to right. The leftmost item appears first in the octet stream and the rightmost item appears last.
- Specified alignment gaps, which can appear in the octet stream before and/or within an item, are not shown in the figures (see Subclause 14.2.2 on page 511).
- Ellipsis points between items labelled "first" and "last" indicate that any number of items can appear in the octet stream. Unless otherwise stated, at least one item, which would be both first and last, must appear. Ellipsis points between items labelled "first" and "penultimate" are used similarly. Figure 14-12 on page 520 shows one example of this notation.
- Braces and arrows indicate an item whose composition is exploded in another part of the figure. Figure 14-17 on page 522 shows one example of this notation.
- "Max" is the abbreviation for "maximum", and "rep" is the abbreviation for "representation".

14.3.2 Arrays

An **array** is an ordered, indexed collection of elements of a single type. The elements of an array can be of any NDR primitive or constructed type except arrays, pipes, conformant structures and context handles.

NDR defines several representations for arrays. The representation used depends on:

- whether the array is uni-dimensional or multi-dimensional
- whether the array is conformant
- whether the array is varying.

NDR defines special representations for arrays of strings (see Subclause 14.3.4 on page 524), for structures that contain some kinds of arrays (see Subclause 14.3.6 on page 526), and for arrays that contain pointers (see Subclause 14.3.11 on page 530).

14.3.2.1 Uni-dimensional Fixed Arrays

A fixed array is an array that is neither conformant nor varying. In a fixed array, the number of elements is known beforehand.

NDR represents a fixed array as an ordered sequence of representations of the array elements.

Figure 14-12 on page 520 illustrates a fixed array as it appears in the octet stream.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

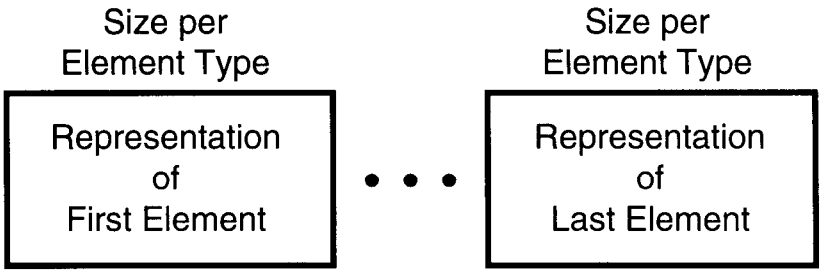


Figure 14-12 Uni-dimensional Fixed Array Representation

14.3.2.2 Uni-dimensional Conformant Arrays

A conformant array is an array in which the maximum number of elements is not known beforehand and therefore is included in the representation of the array.

NDR represents a conformant array as an ordered sequence of representations of the array elements, preceded by an unsigned long integer. The integer gives the number of array elements transmitted, including empty elements.

A conformant array can contain at most $2^{32}-1$ elements.

Figure 14-13 illustrates a conformant array as it appears in the octet stream.

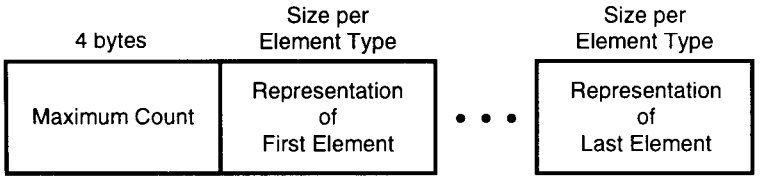


Figure 14-13 Uni-dimensional Conformant Array Representation

14.3.2.3 Uni-dimensional Varying Arrays

A varying array is an array in which the actual number of elements passed in a given call varies and therefore is included in the representation of the array.

NDR represents a varying array as an ordered sequence of representations of the array elements, preceded by two unsigned long integers. The first integer gives the offset from the first index of the array to the first index of the actual subset being passed. The second integer gives the actual number of elements being passed.

A varying array can contain at most $2^{32}-1$ elements.

Figure 14-14 illustrates a varying array as it appears in the octet stream.

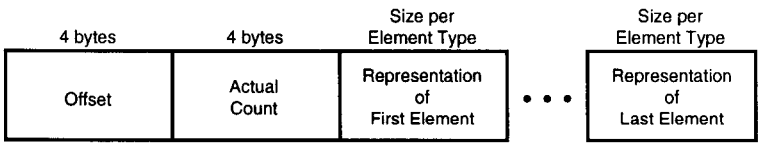


Figure 14-14 Uni-dimensional Varying Array Representation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.2.4 Uni-dimensional Conformant-varying Arrays

An array can be both conformant and varying. Such arrays are called conformant-varying.

NDR represents a conformant and varying array as an ordered sequence of representations of the array elements, preceded by three unsigned long integers. The first integer gives the maximum number of elements in the array. The second integer gives the offset from the first index of the array to the first index of the actual subset being passed. The third integer gives the actual number of elements being passed.

A conformant and varying array can contain at most $2^{32}-1-\mathbf{o}$ elements, where **o** is the offset. The integers that indicate the offset and the actual count are always present, even if the maximum count is 0 (zero).

Figure 14-15 illustrates a conformant and varying array as it appears in the octet stream.

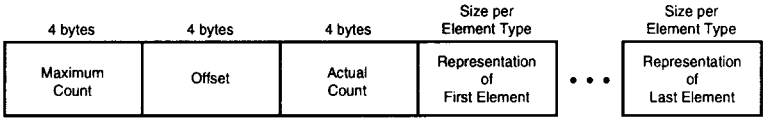


Figure 14-15 Uni-dimensional Conformant and Varying Array Representation

14.3.2.5 Ordering of Elements in Multi-dimensional Arrays

NDR orders multi-dimensional array elements so that the index of the first dimension varies slowest and the index of the last dimension varies fastest.

For example, consider an array **A** in two dimensions with indexes ranging from 0 to 1 in the first dimension and from 0 to 2 in the second dimension. NDR orders the elements of the array as follows:

A(0,0), **A**(0,1), **A**(0,2), **A**(1,0), **A**(1,1),
A(1,2)

where the notation **A(i,j)** denotes the element with index **i** in the first dimension and index **j** in the second dimension of the array **A**.

14.3.2.6 Multi-dimensional Fixed Arrays

A multi-dimensional array is fixed if, in all of its dimensions, the number of elements is known beforehand.

NDR represents fixed multi-dimensional arrays in the same format as fixed uni-dimensional arrays, as shown in Figure 14-16.

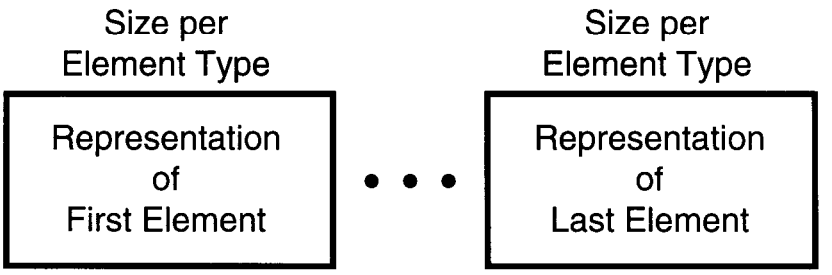


Figure 14-16 Multi-dimensional Fixed Array Representation

14.3.2.7 Multi-dimensional Conformant Arrays

A multi-dimensional array is conformant if the maximum size in any of its dimensions is not known beforehand.

NDR represents a multi-dimensional conformant array as an ordered sequence of unsigned long integers, followed by an ordered sequence of representations of the array elements. The sequence of integers give the maximum size in each dimension of the array.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A multi-dimensional conformant array can span at most $2^{32}-1$ elements in each dimension.

Figure 14-17 illustrates a multi-dimensional conformant array as it appears in the octet stream.

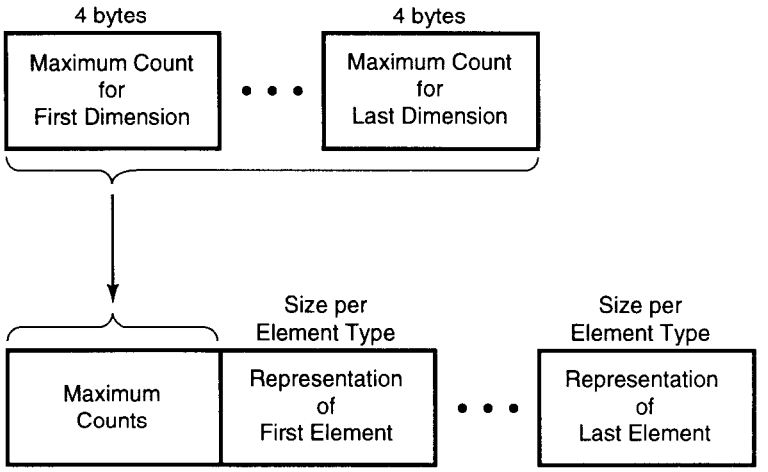


Figure 14-17 Multi-dimensional Conformant Array Representation

14.3.2.8 Multi-dimensional Varying Arrays

A multi-dimensional array is varying if the actual size in any of its dimensions varies.

NDR represents a multi-dimensional varying array as an ordered sequence of pairs of unsigned long integers, followed by an ordered sequence of representations of the array elements. There is one integer pair for each dimension of the array. The first integer of each pair gives the offset from the first index in the dimension to the first index of the subset being passed. The second integer of each pair gives the actual size in the dimension for the subset being passed.

A multi-dimensional varying array can span at most $2^{32}-1$ elements in each dimension.

Figure 14-18 illustrates a multi-dimensional varying array as it appears in the octet stream.

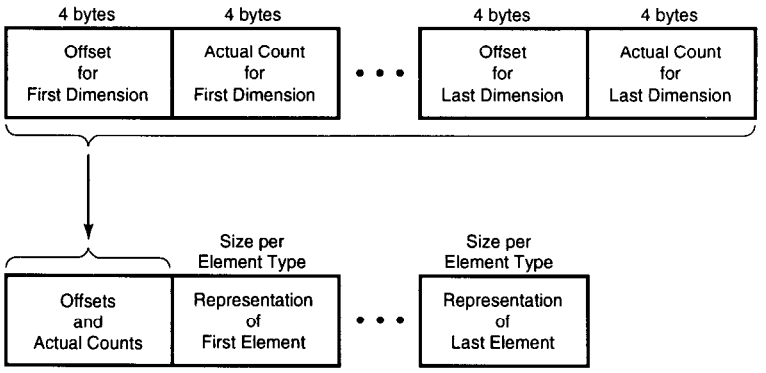


Figure 14-18 Multi-dimensional Varying Array Representation

14.3.2.9 Multi-dimensional Conformant and Varying Arrays

A multi-dimensional array can be both conformant and varying.

NDR represents a multi-dimensional conformant and varying array as an ordered sequence of unsigned long integers, followed by an ordered sequence of pairs of unsigned long integers, followed by an ordered sequence of representations of the array elements. In the sequence of integers, there is one integer for each dimension of the array, and the integers give the maximum size in each dimension. In the sequence of pairs of integers, there is one pair of integers for each dimension of the array; the first integer of each pair gives

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

the offset from the first index in the dimension to the first index of the subset being passed, and the second integer of each pair gives the actual size in the dimension for the subset being passed.

Each dimension of a multi-dimensional conformant and varying array can span at most $2^{32}-1-\mathbf{o}$ elements, where **o** is the offset in that dimension. The integers that indicate the offsets and the actual counts are always present, even if one or more of the maximum counts is 0 (zero).

Figure 14-19 illustrates a multi-dimensional conformant and varying array as it appears in the octet stream.

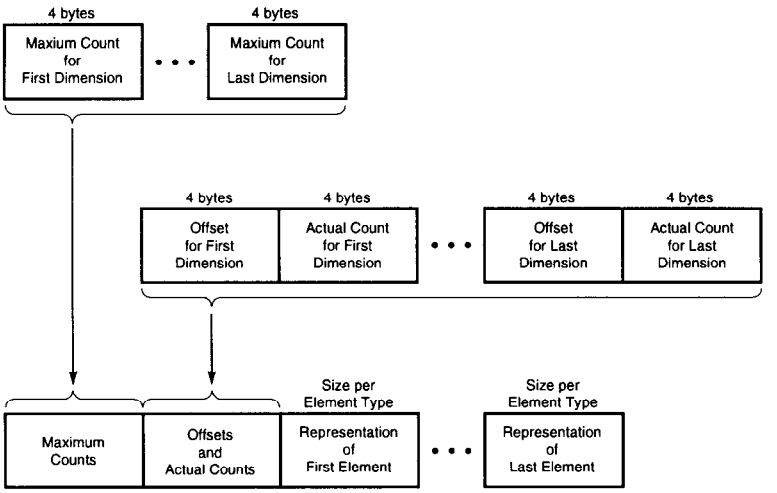


Figure 14-19 Multi-dimensional Conformant and Varying Array Representation

14.3.3 Strings

A string is an indexed or unindexed ordered collection of elements of the same type. The elements in a string must be characters, octets or structures all of whose elements are octets. The actual number of elements passed in a given call varies and therefore is included in the representation of the string.

The last element of a string is a terminator of the same size as the other elements. If the string element size is one octet, the terminator is a 0 (zero) octet. The terminator for a string of multi-byte characters must have 0 (zero) in all its bytes.

Strings can be varying or conformant and varying.

14.3.3.1 Varying Strings

NDR represents a varying string as an ordered sequence of representations of the string elements, preceded by two unsigned long integers. The first integer gives the offset from the first index of the string to the first index of the actual subset being passed. The second integer gives the actual number of elements being passed, including the terminator.

A varying string can contain at most $2^{32}-1$ elements and must contain at least one element, the terminator.

Figure 14-20 illustrates a varying string as it appears in the octet stream.

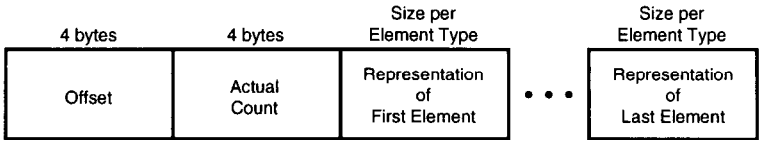


Figure 14-20 Varying String Representation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.3.2 Conformant and Varying Strings

A conformant and varying string is a string in which the maximum number of elements is not known beforehand and therefore is included in the representation of the string.

NDR represents a conformant and varying string as an ordered sequence of representations of the string elements, preceded by three unsigned long integers. The first integer gives the maximum number of elements in the string, including the terminator. The second integer gives the offset from the first index of the string to the first index of the actual subset being passed. The third integer gives the actual number of elements being passed, including the terminator.

A conformant and varying string can contain at most $2^{32}-1-\mathbf{o}$ elements, where **o** is the offset, and must contain at least one element, the terminator.

Figure 14-21 illustrates a conformant and varying string as it appears in the octet stream.

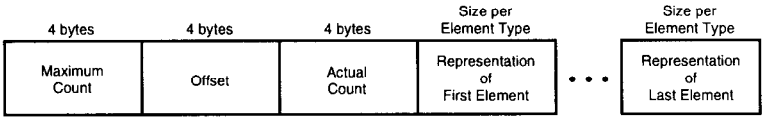


Figure 14-21 Conformant and Varying String Representation

14.3.4 Arrays of Strings

NDR defines a special representation for an array whose elements are strings.

In the NDR representation of an array of strings, any conformance information (maximum element counts) for the strings is removed from the string representations and included in the conformance information for the array, but any variance information (offsets and actual element counts) for the strings remains with the string representations.

NDR requires that all of the strings in an array of strings have the same maximum element count. In the representation of the array that has conformance information, the maximum element count for the strings appears only once, following the maximum element counts for the array.

Figure 14-22 on page 525 illustrates a multi-dimensional conformant and varying array of strings as it appears in the octet stream.

An array of strings can have a degenerate form of the representation in Figure 14-22 on page 525, depending on the properties of the array and of the strings, as follows:

- If the strings are conformant or if any dimension of the array is conformant, then the representation contains maximum element counts for all dimensions of the array and for the strings.
- If the strings are non-conformant and the array is non-conformant, then the representation does not contain any maximum element counts.
- If any dimension of the array is varying, then the representation contains offsets and actual counts for all dimensions of the array.
- If the array is non-varying, then the representation does not contain any offsets or actual counts for the array, although it does contain offsets and actual counts for the strings.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

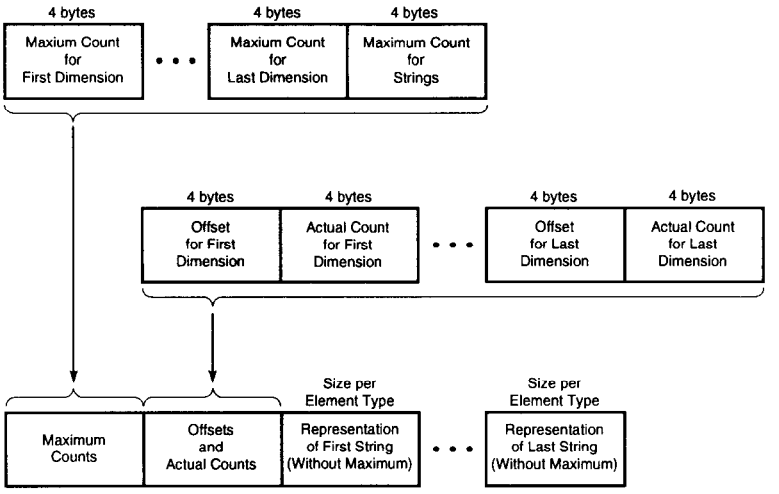


Figure 14-22 Multi-dimensional Conformant and Varying Array of Strings

14.3.5 Structures

A structure is an ordered collection of members, not necessarily all of the same type. A structure member can be of any NDR primitive or constructed type. However, a conformant array can appear in a structure only as the last member, and a structure that contains a conformant array can appear in another structure only as the last member.

NDR represents a structure as an ordered sequence of representations of the structure members.

Figure 14-23 illustrates a structure as it appears in the octet stream.

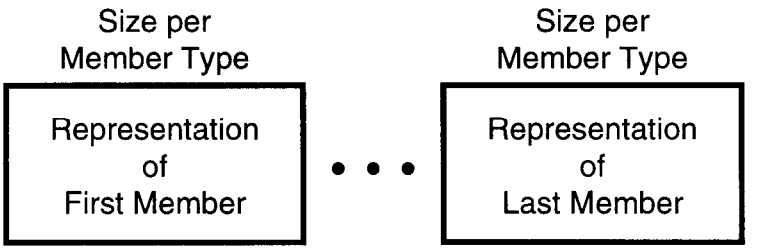


Figure 14-23 Structure Representation

NDR defines special representations for structures that contain some kinds of arrays (see Subclause 14.3.6 on page 526) and for structures that contain pointers (see Subclause 14.3.11 on page 530).

14.3.5.1 Alignment of Constructed Types

NDR enforces NDR alignment of structured data. As with primitive data types, an alignment, 'n', is determined for the structure. Where necessary, an alignment gap of octets of unspecified value preceeds the data in the NDR octet stream. This gap is the smallest size sufficient to align the first field of the structure on an NDR octet stream index of 'n'.

The rules for calculating an alignment are:

- If a conformant structure, (that is, a conformant or varying-conformant array, or a structure containing a conformant or varying-conformant array), is embedded in the constructed type, and this is the outermost containing constructed type, (this structure is not contained in another structure), then the size information from the contained conformant structure is positioned so that it preceeds both the containing constructed type (see Subclause 14.3.6 on page 526 for details) and any alignment gap for the constructed type. The size information itself is aligned according to the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

alignment rules for primitive data types. The data of the constructed type is then aligned according to the alignment rules for the constructed type. In other words, the size information preceeds the structure and is aligned independently of the structure alignment.

- The alignment of the structure in the octet stream is the largest of the alignments of the fields it contains. These fields may also be constructed types. The same alignment rules apply recursively to constructed types.
- The fields within the structure are aligned according to the following rules:
  - — scalar primitives are aligned according to Subclause 14.2.2 on page 511.
  - — pointer alignment is always modulo 4.
  - — struct alignment is determined by recursive application of these rules.
  - — array alignment is the largest alignment of the array element type and the size information type, if any.
  - — union alignment is the largest alignment of the union discriminator and all of the union arms.

Note: The above notions of the union alignment and of the array alignment apply only for calculating the NDR alignment of a struct and do not apply to the actual NDR alignment of a union or an array. For example, the NDR alignment of a union is determined by the tag type and the arm actually to be transmitted, not the largest of the union arms. Similarly, the NDR alignment of an array is determined by the element type alignment, which would be the largest arm of the union for the case of an array of unions.

14.3.6 Structures Containing Arrays

NDR defines special representations for a structure that contains a conformant array or a conformant and varying array.

14.3.6.1 Structures Containing a Conformant Array

A structure can contain a conformant array only as its last member.

In the NDR representation of a structure that contains a conformant array, the unsigned long integers that give maximum element counts for dimensions of the array are moved to the beginning of the structure, and the array elements appear in place at the end of the structure. If a structure that contains a conformant array itself a member of another structure, the maximum element counts are further moved to the beginning of the containing structure. This construction iterates through all enclosing structures.

Figure 14-24 illustrates a structure containing a conformant array as it appears in the octet stream.

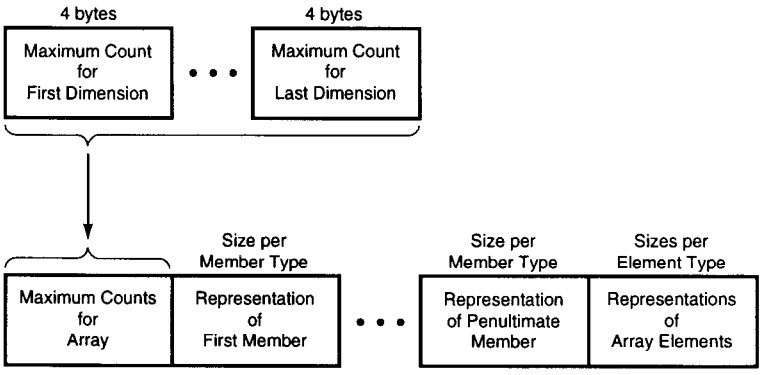


Figure 14-24 Representation of a Structure Containing a Conformant Array

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.6.2 Structures Containing a Conformant and Varying Array

A structure can contain a conformant and varying array only as its last member.

In the NDR representation of a structure that contains a conformant and varying array, the maximum counts for dimensions of the array are moved to the beginning of the structure, but the offsets and actual counts remain in place at the end of the structure, immediately preceding the array elements. If a structure that contains a conformant and varying array is itself a member of another structure, the maximum counts are further moved to the beginning of the containing structure. This construction iterates through all enclosing structures.

Figure 14-25 illustrates a structure containing a conformant and varying array as it appears in the octet stream. The offsets and actual counts iterate pairwise.

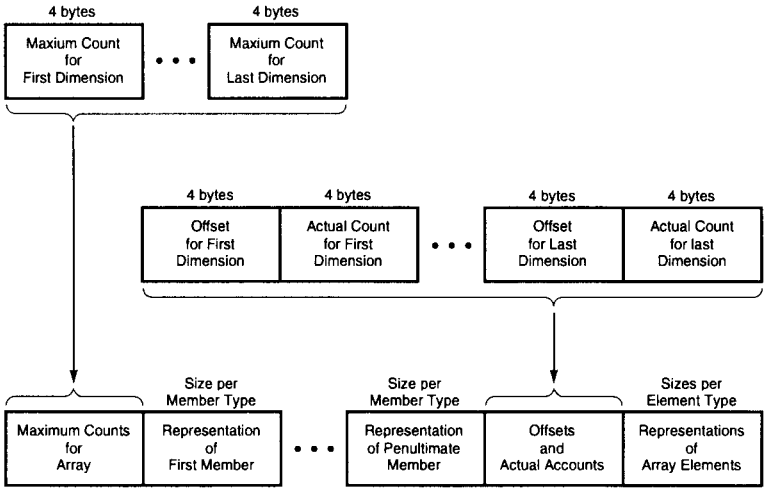


Figure 14-25 Representation of a Structure Containing a Conformant and Varying Array

14.3.7 Unions

A union is a collection of members, not necessarily of the same type, from which one member is selected in any given instance by a discriminating tag. A union member can be of any NDR primitive or constructed type except pipes. A union tag can be of any NDR integer, character or Boolean type.

NDR represents a union as a representation of the tag followed by a representation of the selected member.

Figure 14-26 illustrates a union as it appears in the octet stream.

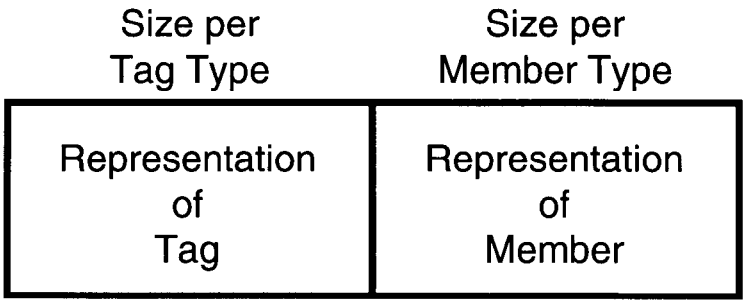


Figure 14-26 Union Representation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8 Pipes

A pipe is an ordered sequence of elements, all of the same type; the number of elements in a pipe is determined dynamically and is potentially unlimited. Elements in a pipe can be of any NDR primitive or constructed type except pipes, conformant and/or varying arrays, and structures that contain conformant and/or varying arrays.

NDR represents a pipe as a sequence of chunks, not necessarily all containing the same number of elements. A chunk can contain at most $2^{32}-1$ elements of the pipe. The number of chunks is potentially unlimited. NDR represents each chunk as an ordered sequence of representations of the elements in the chunk, preceded by an unsigned long integer giving the number of elements in the chunk. The final chunk is empty; it contains no elements and consists only of an unsigned long integer with the value 0 (zero).

The NDR representation of a pipe can be regarded as a sequence of representations of one-dimensional conformant arrays, of length>0, terminated by a zero-length array.

A pipe cannot be an element of another pipe, an element of an array, a member of a structure or variant structure, or a member of a union.

Figure 14-27 illustrates a pipe as it appears in the octet stream.

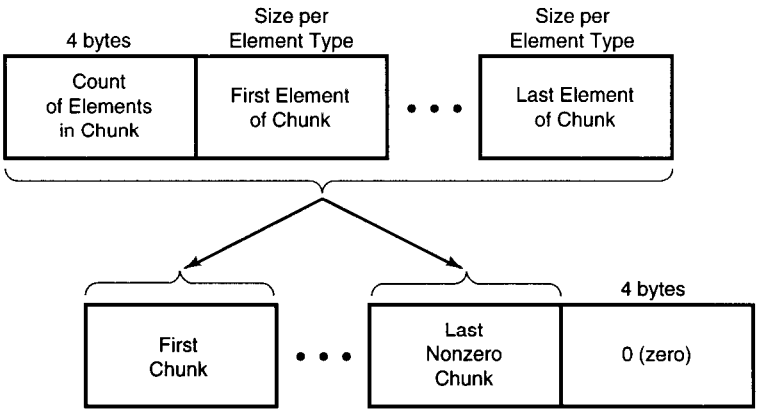


Figure 14-27 Pipe Representation

14.3.9 Pointers

NDR defines two classes of pointers that differ both in semantics and in representation:

- **reference pointers**, which cannot be null and cannot be aliases
- **full pointers**, which can be null and can be an aliases.

If a pointer points to nothing, it is null.

If the input and output octet streams pertaining to one remote procedure call contain several pointers that point to the same thing, the first of these pointers to be transmitted is considered primary and the others are considered aliases.

The scope of aliasing for a pointer extends to all streams transmitted in the service of one remote procedure call; that is, any inputs in the request that initiates the call, and any outputs in the response to the call.

Aliasing does not apply to null pointers.

We refer to pointers that are parameters in remote procedure calls as **top-level pointers** and we refer to pointers that are elements of arrays, members of structures, or members of unions as **embedded pointers**. NDR defines different representations for top-level and embedded pointers. Subclause 14.3.10 on page 529 describes the NDR representation for top-level pointers. Subclause 14.3.11 on page 530 describes the NDR representation for embedded pointers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.10 Top-level Pointers

The following subclauses describe the NDR representation for pointers that are parameters in remote procedure calls.

14.3.10.1 Top-level Full Pointers

NDR represents a null full pointer as an unsigned long integer with the value 0 (zero).

NDR represents the first instance in a octet stream of a non-null full pointer in two parts: the first part is a non-zero unsigned long integer that identifies the referent; the second part is the representation of the referent. NDR represents subsequent instances in the same octet stream of the same pointer only by the referent identifier.

Each referent in the input and output streams pertaining to one remote procedure call is associated with a referent identifier. A primary pointer and its aliases all have the same referent identifier.

On input to the call, if there are **n** distinct referents of full pointers, the **n** referent identifiers are chosen from the set of integers 1, . . ., **n**. On output from the call, if there are **m** new distinct referents, the referent identifiers for the new referents are chosen from the set **n**+1, . . .,**n**+**m**. Similar additions to the set of referent identifiers can also be made at each callback that occurs within the execution of the call.

These requirements extend to embedded full pointers as well.

Figure 14-28 illustrates the three possible representations for top-level full pointers.

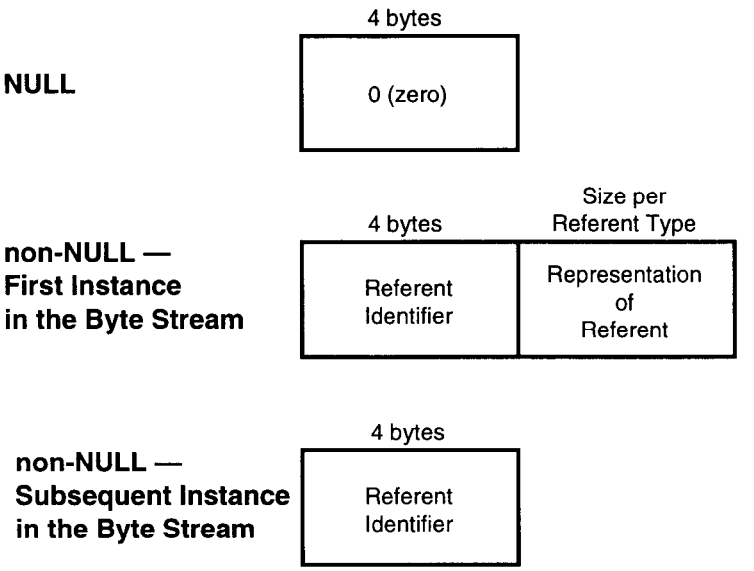


Figure 14-28 Top-level Full Pointer Representation

14.3.10.2 Top-level Reference Pointers

A reference pointer cannot be null; it must point to a referent.

NDR represents a top-level reference pointer simply as the representation of its referent.

Figure 14-29 on page 530 illustrates a top-level reference pointer as it appears in the octet stream.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

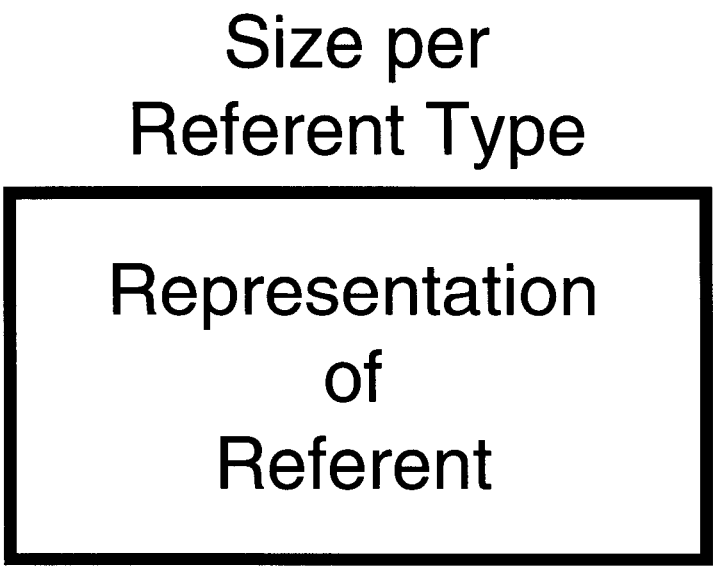


Figure 14-29 Top-level Reference Pointer Representation

14.3.11 Embedded Pointers

The following subclauses describe the NDR representation for pointers that are elements of arrays, members of structures or members of unions.

In the NDR representation of an embedded pointer, the representation of the pointer referent is sometimes deferred to a later position in the octet stream, while the pointer itself is always represented in place as part of the constructed type.

14.3.11.1 Embedded Full Pointers

An embedded full pointer is represented by an unsigned long integer. If the pointer is null, the integer has the value 0 (zero). If the pointer is non-null, the integer is the referent identifier.

The representation of the referent of a primary pointer may be deferred to a later position in the octet stream. Subclause 14.3.11.3 on page 531 describes the algorithm for deferral. Except for this possible deferral, the representation of an embedded full pointer is identical to that of a top-level full pointer.

Figure 14-30 on page 531 illustrates the three possible representations for embedded full pointers.

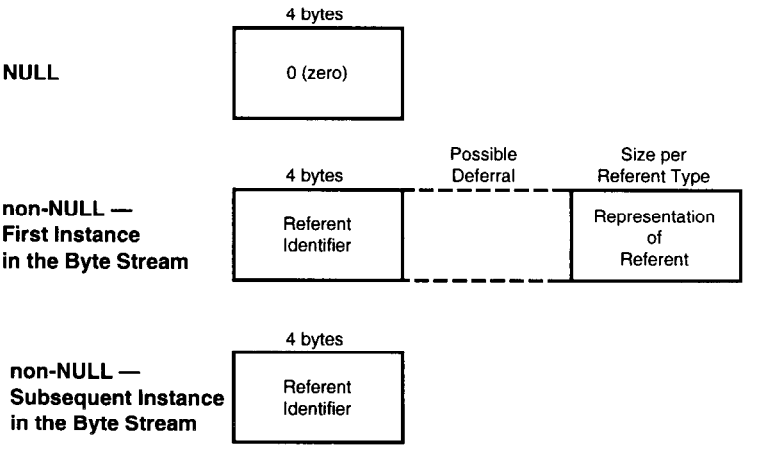


Figure 14-30 Embedded Full Pointer Representations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.11.2 Embedded Reference Pointers

An embedded reference pointer is represented in two parts, a 4 octet value in place and a possibly deferred representation of the referent.

The reference pointer itself is represented by 4 octets of unspecified value. The four octets are aligned as if they were a long integer.

The special case of an array of reference pointers embedded in a structure has no NDR representation, that is; there is no 4-byte unspecified value transmitted.

The representation of the referent of the reference pointer may be deferred to a later position in the octet stream. Subclause 14.3.11.3 describes the algorithm for deferral.

Figure 14-31 illustrates an embedded reference pointer as it appears in the octet stream.

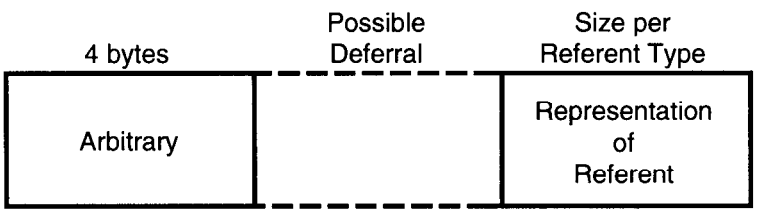


Figure 14-31 Embedded Reference Pointer Representation

14.3.11.3 Algorithm for Deferral of Referents

If a pointer is embedded in an array, structure or union, the representation of its referent is deferred to a position in the octet stream that follows the representation of the construction that embeds the pointer. Representations of pointer referents are ordered according to a left-to-right, depth-first traversal of the embedding construction. Following is an elaboration of the deferral algorithm in detail:

- If an array, structure, or union embeds a pointer, the representation of the referent of the pointer is deferred to a position in the octet stream that follows the representation of the embedding construction.
- If an array or structure embeds more than one pointer, all pointer referent representations are deferred, and the order in which referents are represented is the order in which their pointers appear in place in the array or structure.
- If an array, structure or union embeds another array, structure or union, referent representations for the embedded construction are further deferred to a position in the octet stream that follows the representation of the embedding construction. The set of referent representations for the embedded construction is inserted among the referent representations for any pointers in the embedding construction, according to the order of elements or members in the embedding construction.
- The deferral of referent representations iterates through all successive embedding arrays, structures, and unions to the outermost array, structure or union.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.4 NDR Input and Output Streams

NDR represents the set of inputs or outputs in a remote procedure call as a octet stream. The octet stream consists of two parts: one part represents data that are pipes and the other part represents data that are not pipes. In the representation of a set of inputs, the part representing pipes appears last. In the representation of a set of outputs, the part representing pipes appears first.

Each part of the octet stream is aligned at an octet stream index that is a multiple of 8. To produce this alignment, a gap of octets of unspecified value may separate the two parts. The figures in this subclause do not show such gaps.

If an operation returns a result, the representation of the result appears after all parameters in the output stream.

Figure 14-32 illustrates the octet stream that represents a set of inputs.

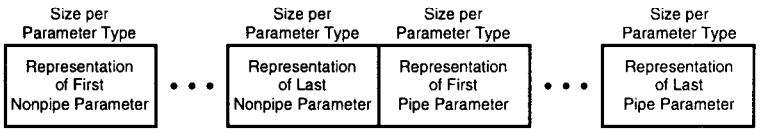


Figure 14-32 NDR Input Stream

Figure 14-33 illustrates the octet stream that represents a set of outputs.

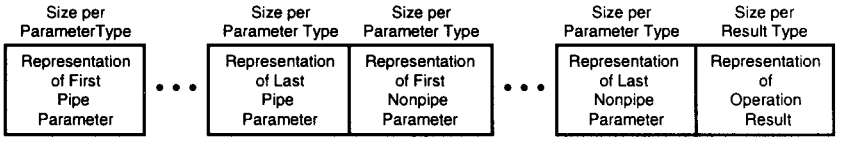


Figure 14-33 NDR Output Stream

The octet stream representing a set of inputs or outputs is transmitted either as the body of one PDU or as the bodies of several PDUs, as described in Section 12.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A.
(normative)
Universal Unique Identifier

This Annex specifies the syntax and semantics of Universal Unique Identifiers (UUIDs).

A UUID is an identifier that is unique across both space and time[6], with respect to the space of all UUIDs. A UUID can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects across a network.

The generation of UUIDs does not require a registration authority for each single identifier. Instead, it requires a unique value over space for each UUID generator. This spatially unique value is specified as an IEEE 802 address, which is usually already applied to network-connected systems. This 48-bit address can be assigned based on an address block obtained through the IEEE registration authority. This UUID specification assumes the availability of an IEEE 802 address.

6. To be precise, the UUID consists of a finite bit space. Thus the time value used for constructing a UUID is limited and will roll over in the future (approximately at A.D. 3400, based on the specified algorithm).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.1 Format

Table A-1 shows the format of a UUID.

Table A-1 UUID Format

Field	NDR Data Type	Octet #	Note
time_low	**unsigned long**	0-3	The low field of the timestamp.
time_mid	**unsigned short**	4-5	The middle field of the timestamp.
time_hi_and_version	**unsigned short**	6-7	The high field of the timestamp multiplexed with the version number.
clock_seq_hi_and_reserved	**unsigned small**	8	The high field of the clock sequence multiplexed with the variant.
clock_seq_low	**unsigned small**	9	The low field of the clock sequence.
node	**character**	10-15	The spatially unique node identifier.

The UUID consists of a record of 16 octets and must not contain padding between fields. The total size is 128 bits.

To minimise confusion about bit assignments within octets, the UUID record definition is defined only in terms of fields that are integral numbers of octets. The version number is multiplexed with the time stamp (**time_high**), and the variant field is multiplexed with the clock sequence (**clock_seq_high**).

The timestamp is a 60 bit value. For UUID version 1, this is represented by Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar).

The version number is multiplexed in the 4 most significant bits of the **time_hi_and_version** field. Table A-2 lists currently defined versions of the UUID.

Table A-2 UUID version Field

Msb1	Msb2	Msb3	Msb4	Version	Description
0	0	0	1	1	DCE version, as specified herein.
0	0	1	0	2	DCE Security version, with embedded POSIX UIDs.

The variant field determines the layout of the UUID. The structure of DCE UUIDs is fixed across different versions. Other UUID variants may not interoperate with DCE UUIDs. Interoperability of UUIDs is defined as the applicability of operations such as string conversion, comparison, and lexical ordering across different systems. The **variant** field consists of a variable number of the msbs of the **clock_seq_hi_and_reserved** field. Table A-3 lists the contents of the DCE variant field.

Table A-3 UUID variant Field

Msb1	Msb2	Msb3	Description
0	—	—	Reserved, NCS backward compatibility.
1	0	—	DCE variant.
1	1	0	Reserved, Microsoft Corporation GUID.
1	1	1	Reserved for future definition.

The clock sequence is required to detect potential losses of monotonicity of the clock. Thus, this value marks discontinuities and prevents duplicates. An algorithm for generating this value is outlined in Subclause A.2.1 on page 536. The clock sequence is encoded in the 6 least significant bits of the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

clock_seq_hi_and_reserved field and in the **clock_seq_low** field.

The **node** field consists of the IEEE address, usually the host address. For systems with multiple IEEE 802 nodes, any available node address can be used. The lowest addressed octet (octet number 10) contains the global/local bit and the unicast/multicast bit, and is the first octet of the address transmitted on an 802.3 LAN.

Depending on the network data representation, the multi-octet unsigned integer fields are subject to byte swapping when communicated between different endian machines.

The nil UUID is special form of UUID that is specified to have all 128 bits set to 0 (zero).

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.2 Algorithms for Creating a UUID

Various aspects of the algorithm for creating a UUID are discussed in the following subclauses. UUID generation requires a guarantee of uniqueness within the node ID for a given variant and version. Interoperability is provided by complying with the specified data structure. To prevent possible UUID collisions, which could be caused by different implementations on the same node, compliance with the algorithms specified here is required.

A.2.1 Clock Sequence

The clock sequence value must be changed whenever:

- The UUID generator detects that the local value of UTC has gone backward; this may be due to normal functioning of the DCE Time Service.
- The UUID generator has lost its state of the last value of UTC used, indicating that time **may** have gone backward; this is typically the case on reboot.

While a node is operational, the UUID service always saves the last UTC used to create a UUID. Each time a new UUID is created, the current **UTC** is compared to the saved value and if either the current value is less (the non-monotonic clock case) or the saved value was lost, then the **clock sequence** is incremented modulo 16,384, thus avoiding production of duplicate UUIDs.

The **clock sequence** must be initialised to a random number to minimise the correlation across systems. This provides maximum protection against **node** identifiers that may move or switch from system to system rapidly. The initial value **shall not** be correlated to the node identifier.

The rule of initialising the **clock sequence** to a random value is waived if, and only if all of the following are true:

a) The **clock sequence** value is stored in a form of non-volatile storage.

b) The system is manufactured such that the IEEE address ROM is designed to be inseparable from the system by either the user or field service, so that it cannot be moved to another system.

c) The manufacturing process guarantees that only new IEEE address ROMs are used.

d) Any field service, remanufacturing or rebuilding process that could change the value of the clock sequence must reinitialise it to a random value.

In other words, the system constraints prevent duplicates caused by possible migration of the IEEE address, while the operational system itself can protect against non-monotonic clocks, except in the case of field service intervention. At manufacturing time, such a system may initialise the clock sequence to any convenient value.

A.2.2 System Reboot

There are two possibilities when rebooting a system:

a) the UUID generator state — the last UTC, adjustment, and clock sequence — of the UUID service has been restored from non-volatile store

b) the state of the last UTC or adjustment has been lost.

If the state variables have been restored, the UUID generator just continues as normal. Alternatively, if the state variables cannot be restored, they are reinitialised, and the clock sequence is changed. If the clock sequence is stored in non-volatile store, it is incremented; otherwise, it is reinitialised to a new random value.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A.2.3 Clock Adjustment

UUIDs may be created at a rate greater than the system clock resolution. Therefore, the system must also maintain an adjustment value to be added to the lower-order bits of the time. Logically, each time the system clock ticks, the adjustment value is cleared. Every time a UUID is generated, the current adjustment value is read and incremented atomically, then added to the UTC time field of the UUID.

A.2.4 Clock Overrun

The 100 nanosecond granularity of time should prove sufficient even for bursts of UUID creation in the next generation of high-performance multiprocessors. If a system overruns the clock adjustment by requesting too many UUIDs within a single system clock tick, the UUID service may raise an exception, handled in a system or process-dependent manner either by:

- terminating the requester
- reissuing the request until it succeeds
- stalling the UUID generator until the system clock catches up.

If the processors overrun the UUID generation frequently, additional node identifiers and clocks may need to be added.

A.2.5 UUID Generation

UUIDs are generated according to the following algorithm:

a) Determine the values for the UTC-based timestamp and clock sequence to be used in the UUID. Subclause A.1 on page 534 and Subclause A.2.1 on page 536 define how to determine these values. For the purposes of this algorithm, consider the timestamp to be a 60-bit unsigned integer and the clock sequence to be a 14-bit unsigned integer. Sequentially number the bits in a field, starting from 0 (zero) for the least significant bit.

b) Set the **time_low** field equal to the least significant 32-bits (bits numbered 0 to 31 inclusive) of the time stamp in the same order of significance.

c) Set the **time_mid** field equal to the bits numbered 32 to 47 inclusive of the time stamp in the same order of significance.

d) Set the 12 least significant bits (bits numbered 0 to 11 inclusive) of the **time_hi_and_version** field equal to the bits numbered 48 to 59 inclusive of the time stamp in the same order of significance.

e) Set the 4 most significant bits (bits numbered 12 to 15 inclusive) of the **time_hi_and_version** field to the 4-bit version number, as shown in Table A-2 on page 534.

f) Set the **clock_seq_low** field to the 8 least significant bits (bits numbered 0 to 7 inclusive) of the **clock sequence** in the same order of significance.

g) Set the 4 least significant bits (bits numbered 0 to 3 inclusive) of the **clock_seq_hi_and_reserved** field to the 4 most significant bits (bits numbered 8 to 13 inclusive) of the **clock sequence** in the same order of significance.

h) Set the 4 most significant bits (bits numbered 4 to 7 inclusive) of the **clock_seq_hi_and_reserved** field as shown in Table A-4.

Table A-4 The 4 Msb of clock_seq_hi_and_reserved

Bit 15	Bit 14	Bit 13	Bit 12
1	0	0	0

i) Set the **node** field to the 48-bit IEEE address in the same order of significance as the address.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.3 String Representation of UUIDs

For use in human readable text, a UUID string representation is specified as a sequence of fields, some of which are separated by single dashes.

Each field is treated as an integer and has its value printed as a zero-filled hexadecimal digit string with the most significant digit first. The hexadecimal values a to f inclusive are output as lower case characters, and are case insensitive on input. The sequence is the same as the UUID constructed type.

The formal definition of the UUID string representation is provided by the following extended BNF:

```
UUID                    = <time_low> <hyphen> <time_mid> <hyphen>
                          <time_high_and_version> <hyphen>
                          <clock_seq_and_reserved>
                          <clock_seq_low> <hyphen> <node>
time_low                = <hexOctet> <hexOctet> <hexOctet> <hexOctet>
time_mid                = <hexOctet> <hexOctet>
time_high_and_version   = <hexOctet> <hexOctet>
clock_seq_and_reserved = <hexOctet>
clock_seq_low           = <hexOctet>
node                    = <hexOctet><hexOctet><hexOctet>
                          <hexOctet><hexOctet><hexOctet>
hexOctet                = <hexDigit> <hexDigit>
hexDigit                = <digit> | <a> | <b> | <c> | <d> | <e> | <f>
digit                   = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                          "8" | "9"
hyphen                  = "-"
a                       = "a" | "A"
b                       = "b" | "B"
c                       = "c" | "C"
d                       = "d" | "D"
e                       = "e" | "E"
f                       = "f" | "F"
```

The following is an example of the string representation of a UUID:

```
2fac1234-31f8-11b4-a222-08002b34c003
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.4 Comparing UUIDs

Table A-5 lists the UUID fields in order of significance, from most significant to least significant, for purposes of UUID comparison. The table also shows the data types applicable to the fields.

Table A-5 Field Order and Type

Field	Type
time_low	Unsigned 32-bit integer
time_mid	Unsigned 16-bit integer
time_hi_and_version	Unsigned 16-bit integer
clock_seq_hi_and_reserved	Unsigned 8-bit integer
clock_seq_low	Unsigned 8-bit integer
node	Unsigned 48-bit integer

Consider each field to be an unsigned integer as shown in Table A-5. To compare a pair of UUIDs, arithmetically compare the corresponding fields from each UUID in order of significance and according to their data type. Two UUIDs are equal if and only if all the corresponding fields are equal. The first of two UUIDs follows the second if the most significant field in which the UUIDs differ is greater for the first UUID. The first of a pair of UUIDs precedes the second if the most significant field in which the UUIDs differ is greater for the second UUID.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex B.
(normative)
Protocol Sequence Strings

This Annex lists valid RPC protocol sequence strings. These can be used with routines that take a protocol sequence string argument (type **unsigned_char_t***). They can also be used to construct the **rpc-protocol-sequence** portion of a string binding. (See Subclause 3.1 on page 47 for more information on protocol sequence strings and string bindings.)

Table B-1 shows the valid RPC protocol sequence strings.

Table B-1 RPC Protocol Sequence Strings

String	Description
"ncacn_ip_tcp"	CO over Internet Protocol: Transmission Control Protocol (TCP/IP)
"ncacn_dnet_nsp"	CO over DECnet: Network Services Protocol (DECnet Phase IV)
"ncacn_osi_dna"	CO over Open Systems Interconnection (DECnet Phase V)
"ip"	CL over Internet Protocol: User Datagram Protocol (UDP/IP)
"ncadg_ip_udp"	
"dds"	CL over Domain Datagram Service
"ncadg_dds"	

To ensure interoperation, applications should use only these strings. Not all implementations support all the protocol sequences listed. Applications can use a particular protocol sequence only if the implementation supports that sequence.

Subclause 3.1 on page 47 shows how to construct the network address and endpoint portions of a string binding for the TCP/IP and UDP/IP protocols. Contact OSF for information on constructing string bindings for the other protocol sequences or to obtain protocol string assignments. To avoid conflicts or multiple strings for a protocol sequence, all protocol sequence strings must be registered with OSF.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex C.
(normative)
Name Syntax Constants

Table C-1 lists defined constant values that specify name syntax. These can be used with RPC routines that take a **name_syntax** argument.

Table C-1 RPC Name Syntax Defined Constants

Constant	Value	Description
rpc_c_ns_syntax_default	0	Default syntax
rpc_c_ns_syntax_dce	3	OSF DCE name syntax

When the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable is defined, its value determines the default syntax. When it is not defined, the RPC run-time system uses the rpc_c_ns_syntax_dce name syntax as the default. The syntaxes that correspond to these values will be specified in the **DCE: Directory Services** specification.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

This page intentionally left blank.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex D.
(normative)
Authentication, Authorisation and Protection-level Arguments

This Annex lists possible values for several arguments used by authentication-related RPC routines. The RPC API authentication-related routines are designed to be authentication and authorisation service-independent, but the values taken by some arguments to these routines are necessarily service-specific. The ISO C standard currently specifies some authentication and authorisation services, but implementations may use others, and the ISO C standard may in the future specify others. This Annex specifies argument values that are applicable to the services currently specified in the ISO C standard. The use of these values assures portability of applications to implementations that used the ISO C standard specified services.

D.1 The authn_svc Argument

The **authn_svc** argument is used to specify an authentication service. The following list gives the possible values for currently specified authentication services.

rpc_c_authn_none
: No authentication.

rpc_c_authn_dce_secret
: DCE shared-secret key authentication.

rpc_c_authn_default
: DCE default authentication service (should equal one of the other defined values).

D.2 The authz_svc Argument

The **authz_svc** argument is used to specify an authorisation service. The following list gives the possible values for currently specified authorisation services:

rpc_c_authz_none
: The server performs no authorisation. This is valid only if the **authn_svc** argument is rpc_c_authn_none.

rpc_c_authz_name
: The server performs authorisation based on the client principal name.

rpc_c_authz_dce
: The server performs authorisation using the client's DCE Privilege Attribute Certificate (PAC), which is sent to the server with each remote procedure call made with a given binding. Typically, access is checked against DCE Access Control Lists (ACLs).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

D.3 The protect_level Argument

The **protect_level** argument is used to specify which level of protection to apply to authenticated RPC communications. The following list gives possible values for this argument:

rpc_c_protect_level_default
: Use the default protection level for the specified authentication service.

rpc_c_protect_level_none
: Perform no protection.

rpc_c_protect_level_connect
: The client and server identities are exchanged and cryptographically verified. Strong mutual authentication is achieved on the connection and is protected against replays. There are no protection services per PDU.

rpc_c_protect_level_call
: This level offers the rpc_c_protect_level_connect services, plus integrity protection on the first fragment of each call only. This level is currently not supported by the protocol. Any request for this level will be automatically upgraded to rpc_c_protect_level_pkt.

rpc_c_protect_level_pkt
: This level offers the rpc_c_protect_level_connect services, plus detection of misordering or replay of PDUs. There is no protection against PDU modification.

rpc_c_protect_level_pkt_integrity
: This level offers the rpc_c_protect_level_pkt services, plus detection of PDU modification.

rpc_c_protect_level_pkt_privacy
: This level offers the rpc_c_protect_level_pkt_integrity services, plus privacy of stub call arguments. Run-time and lower-layer headers are not protected by these services.

The protection-level values are listed in canonical order from the lowest to highest level of protection. However, except for the first two levels, the actual definition of each level depends on the underlying protocol, and not all levels may be provided by all protocols.

When an application calls the **rpc_set_auth_info** routine with a protection level that is not supported, the RPC run-time system attempts to upgrade the request to the next highest supported level.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

D.4 The privs Argument

The **privs** argument returns a handle to the authorisation or privilege information for a client binding handle. An application must cast this value to an appropriate type for the authorisation service in use. Table D-1 shows the appropriate casts for supported authorisation services:

Table D-1 Casts for Authorisation Information

For authz_svc value:	privs contains this data:	Use this cast:
rpc_authz_none	A NULL value.	None
rpc_authz_name	The calling client's principal name.	**(unsigned_char_t *)**
rpc_authz_dce	The calling client's privilege attribute certificate.	**(sec_id_pac_t *)**

D.5 The server_princ_name Argument

The **server_princ_name** argument specifies a server principal name. The syntax of this name depends on the authentication service in use. This syntax will be specified in the **DCE: Security Services** specification.

D.6 The auth_identity Argument

The **auth_identity** argument specifies an application's authentication and authorisation credentials.

When using the rpc_c_authn_dce_secret authentication service and any authorisation service, this value must be a **sec_login_handle_t**.

D.7 Key Functions

When a server application calls **rpc_server_register_auth_info**() to register authentication information with the RPC run-time system, it specifies an authentication service, using the **authn_svc** argument. It may also specify a server-provided key acquisition function, using the **get_key_fn** argument. To specify the default key acquisition function for the authentication service specified by **authn_svc**, the application supplies NULL for the **get_key_fn** argument. The application may also supply an argument to be passed to the key acquisition function, using the **arg** argument.

The values of these arguments determine how the RPC run-time system behaves when it needs to acquire a key for authenticated RPC. Table D-2 on page 548 shows the RPC run-time system behaviour for the supported authentication services.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table D-2 RPC Key Acquisition for Authentication Services

authn_svc	*get_key_fn*	*arg*	**Run-time Behaviour**
rpc_c_authn_default	Ignored	NULL	Uses the default method of encryption key acquisition.
rpc_c_authn_default	Ignored	non-NULL	Uses the default method of encryption key acquisition. The specified argument is passed to the default acquisition function.
rpc_c_authn_none	Ignored	ignored	Authentication is not performed.
rpc_c_authn_dce_secret	NULL	NULL	Uses the default method of encryption key acquisition.
rpc_c_authn_dce_secret	NULL	non-NULL	Uses the default method of encryption key acquisition. The specified argument is passed to the default acquisition function.
rpc_c_authn_dce_secret	non-NULL	NULL	Uses the specified encryption key acquisition routine to obtain keys.
rpc_c_authn_dce_secret	non-NULL	non-NULL	Uses the specified encryption key acquisition routine to obtain keys. The specified argument is passed to the acquisition function.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex E.
(normative)
Reject Status Codes and Parameters

This Annex lists reject status codes for RPC PDUs and the error statuses that may returned by a client stub to an application in **comm_status** and **fault_status** parameters.

E.1 Reject Status Codes

Both **reject** and connection-oriented **fault** PDUs contain a 32-bit field that indicates a server's reason for rejecting an RPC call request. This field is encoded as the body data of the **reject** PDU and as the **status** field of the connection-oriented **fault** PDU header. Table E-1 lists the possible values of this field in hexadecimal notation:

Table E-1 Reject Status Codes

Name	Value	Protocol	Meaning
nca_rpc_version_mismatch	1c000008	CO/CL	The server does not support the RPC protocol version specified in the request PDU.
nca_unspec_reject	1c000009	CO,CL	The request is being rejected for unspecified reasons.
nca_s_bad_actid	1c00000A	CL	The server has no state corresponding to the activity identifier in the message.
nca_who_are_you_failed	1c00000b	CL	The Conversation Manager callback failed.
nca_manager_not_entered	1c00000c	CO,CL	The server manager routine has not been entered and executed.
nca_op_rng_error	1c010002	CO,CL	The operation number passed in the request PDU is greater than or equal to the number of operations in the interface.
nca_unk_if	1c010003	CO,CL	The server does not export the requested interface.
nca_wrong_boot_time	1c010006	CL	The server boot time passed in the request PDU does not match the actual server boot time.
nca_s_you_crashed	1c010009	CL	A restarted server called back a client.
nca_proto_error	1c01000b	CO/CL	The RPC client or server protocol has been violated.
nca_out_args_too_big	1c010013	CO,CL	The output parameters of the operation exceed their declared maximum size.
nca_server_too_busy	1c010014	CO,CL	The server is too busy to handle the call.
nca_unsupported_type	1c010017	CO,CL	The server does not implement the requested operation for the type of the requested object.
nca_invalid_pres_context_id	1c00001c	CO	Invalid presentation context ID.
nca_unsupported_authn_level	1c00001d	CO,CL	The server did not support the requested authentication level.
nca_invalid_checksum	1c00001f	CO,CL	Invalid checksum.
nca_invalid_crc	1c000020	CO,CL	Invalid CRC.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Note: A set of **fault** status values is also encoded in both connectionless and connection-oriented **fault** PDU body data. This data is not used by the RPC protocols themselves and is not listed in Table E-1 on page 549. These fault values and the corresponding application level **fault_status** values are given in Table E-3 on page 552.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

E.2 Possible Failures

E.2.1 comm_status Parameter

The following failures normally indicate a communication failure. Such failures do not necessarily indicate a problem in the RPC or application code. All of the failures listed in Table E-1 on page 549 are returned in a **comm_status** parameter or function result when that mechanism is enabled via the ACS. Portable applications should enable this mechanism as described in Subclause 4.3.9 on page 245.

Table E-2 Failures Returned in a comm_status Parameter

Name	Value
rpc_s_access_control_info_inv	16C9A04A
rpc_s_cancel_timeout	16C9A030
rpc_s_comm_failure	16C9A016
rpc_s_connect_closed_by_rem	16C9A04C
rpc_s_connect_no_resources	16C9A044
rpc_s_connect_rejected	16C9A042
rpc_s_connect_timed_out	16C9A041
rpc_s_connection_closed	16C9A036
rpc_s_host_unreachable	16C9A049
rpc_s_fault_remote_comm_failure	16C9A085
rpc_s_loc_connect_aborted	16C9A04B
rpc_s_network_unreachable	16C9A043
rpc_s_no_memory	16C9A012
rpc_s_no_more_bindings	16C9A0B5
rpc_s_no_ns_permission	16C9A0A8
rpc_s_no_rem_endpoint	16C9A047
rpc_s_op_rng_error	16C9A001
rpc_s_rem_host_crashed	16C9A04D
rpc_s_rem_host_down	16C9A048
rpc_s_rem_network_shutdown	16C9A045
rpc_s_too_many_rem_connects	16C9A046
rpc_s_unknown_if	16C9A02C
rpc_s_unsupported_type	16C9A02D
rpc_s_wrong_boot_time	16C9A006

E.2.2 fault_status Parameter

The following failures normally indicate a failure of the remote application code. All of the failures listed in Table E-3 on page 552 are returned in a **fault_status** parameter or function result when that mechanism is enabled via the ACS. Portable applications should enable this mechanism, as described in Subclause 4.3.9 on page 245.

The four columns represent the fault status name, the hexadecimal value of the fault status, the fault PDU name that is associated with this fault, and the hexadecimal value encoded by the fault PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table E-3 Failures Returned in a fault_status Parameter

Fault Status	Status Value	PDU Fault	PDU Fault Value
rpc_s_call_cancelled	16C9A031	nca_s_fault_cancel	1C00000D
rpc_s_fault_addr_error	16C9A074	nca_s_fault_addr_error	1C000002
rpc_s_fault_context_mismatch	16C9A075	nca_s_fault_context_mismatch	1C00001A
rpc_s_fault_fp_div_by_zero	16C9A076	nca_s_fault_fp_div_zero	1C000003
rpc_s_fault_fp_error	16C9A077	nca_s_fault_fp_error	1C00000F
rpc_s_fault_fp_overflow	16C9A078	nca_s_fault_fp_overflow	1C000005
rpc_s_fault_fp_underflow	16C9A079	nca_s_fault_fp_underflow	1C000004
rpc_s_fault_ill_inst	16C9A07A	nca_s_fault_ill_inst	1C00000E
rpc_s_fault_int_div_by_zero	16C9A07B	nca_s_fault_int_div_by_zero	1C000001
rpc_s_fault_int_overflow	16C9A07C	nca_s_fault_int_overflow	1C000010
rpc_s_fault_invalid_bound	16C9A07D	nca_s_fault_invalid_bound	1C000007
rpc_s_fault_invalid_tag	16C9A07E	nca_s_fault_invalid_tag	1C000006
rpc_s_fault_pipe_closed	16C9A07F	nca_s_fault_pipe_closed	1C000015
rpc_s_fault_pipe_comm_error	16C9A080	nca_s_fault_pipe_comm_error	1C000018
rpc_s_fault_pipe_discipline	16C9A081	nca_s_fault_pipe_discipline	1C000017
rpc_s_fault_pipe_empty	16C9A082	nca_s_fault_pipe_empty	1C000014
rpc_s_fault_pipe_memory	16C9A083	nca_s_fault_pipe_memory	1C000019
rpc_s_fault_pipe_order	16C9A084	nca_s_fault_pipe_order	1C000016
rpc_s_fault_remote_no_memory	16C9A086	nca_s_fault_remote_no_memory	1C00001B
rpc_s_fault_unspec	16C9A087		

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex F.
(normative)
IDL to C-language Mappings

This Annex specifies the bindings of IDL data types to NDR data types and to a set of C-language defined data types. It also specifies the mapping of IDL syntax to the C-language syntax of generated stubs.

F.1 Data Type Bindings

The the data type mappings are specified in Table F-1 on page 554. For definitions of the NDR data types, refer to Section 14. Stubs use the C defined data types; to ensure portability, applications should use them as well. The C types shown in the last column of the table are recommended definitions for the C defined types for 32-bit machines.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table F-1 IDL/NDR/C Type Mappings

IDL Type	NDR Type	Defined C Type	C Type
boolean	boolean	idl_boolean	unsigned char
char	character	idl_char	unsigned char
byte	uninterpreted octet	idl_byte	unsigned char
small	small	idl_small_int	char
short	short	idl_short_int	short int
long	long	idl_long_int	long int

hyper	hyper	idl_hyper_int	16- or 32- Bit Machines:
			Big Endian:
			`struct {`
			`  long high;`
			`  unsigned long low;`
			`}`
			Little Endian:
			`struct {`
			`  unsigned long low;`
			`  long high;`
			`}`
			64-Bit Machines:
			`long`

unsigned small	unsigned small	idl_usmall_int	unsigned char
unsigned short	unsigned short	idl_ushort_int	unsigned short int
unsigned long	unsigned long	idl_ulong_int	unsigned long int

unsigned hyper	unsigned hyper	idl_uhyper_int	16 or 32-Bit Machines:
			Big Endian:
			`struct {`
			`  unsigned long high;`
			`  unsigned long low;`
			`}`
			Little Endian:
			`struct {`
			`  unsigned long low;`
			`  unsigned long high;`
			`}`
			64-Bit Machines:
			`unsigned long`

float	float	idl_float	float
double	double	idl_double	double
handle_t	not transmitted	handle_t	void *
error_status_t	unsigned long	idl_ulong_int	unsigned long int
ISO_LATIN_1	uninterpreted octet	ISO_LATIN_1	byte

ISO_MULTI_LINGUAL	(Note 1.)	ISO_MULTI_LINGUAL	`struct{`
			`  byte row;`
			`  byte column;`
			`}`

ISO_UCS	(Note 1.)	ISO_UCS	`struct{`
			`  byte group;`
			`  byte plane;`
			`  byte row;`
			`  byte column;`
			`}`

a) The **ISO_MULTILINGUAL** and **ISO_UCS** types are structures and are NDR encoded as constructed types.

The recommended values for the Boolean constants are specified in Table F-2 on page 555.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table F-2 Recommended Boolean Constant Values

Constant	Value
TRUE	1
FALSE	0

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

F.2 Syntax Mappings

Unless specified otherwise, the code generated for the C language is syntactically identical to the IDL declarations. The following are the mappings from IDL to the C language for which the syntax is not identical to IDL. These mappings use the conventions, notation and productions defined in Section 4.

- The interface header is not mapped to any C-language construct that is visible to users of RPC. (See Section 2 for a discussion of interface handles.)
- The import declaration causes definitions to be imported from other IDL source files. The import declaration itself has no associated C-language mapping.
- All attributes (for example, <type_attribute>, <union_type_switch_attr>, <union_instance_switch_attr>, <usage_attribute>, <xmit_type>, <field_attribute>, <ptr_attr>, <operation_attribute>, <param_attribute>) and their associated **[ ]** (brackets), and punctuation are not mapped to the C language.
- The IDL constant declaration:

```
const <const_type_spec> <Identifier> = <const_exp>
```

  maps to the C language as:

```
#define <Identifier> (<const_exp>)
```

- The IDL non-encapsulated union declaration

```
union [ <tag> ] { <union_case_label_n_e> <union_arm>
        [ <union_case_label_n_e> <union_arm> ] ...
        [ default : <union_arm> ]
      }
```

  maps to the C language as:

```
union [ <tag> ] { <union_arm> [ <union_arm> ]... }
```

  If an IDL <union_arm> is empty (that is, an instance resolves to only a **;** (semicolon)), then no C mapping is generated for that arm.

- The IDL encapsulated union declaration:

```
union [ <tag> ] switch ( <switch_type_spec> <Identifier> ) [<union_name> ]
    { <union_case_label> [ <union_case_label> ] ... <union_arm>
        [ <union_case_label> [ <union_case_label> ] ... <union_arm> ] ...
    }
```

  maps to the C language as:

```
struct [ <tag> ]
   {
      <switch_type_spec> <Identifier> ;
      union { <union_arm> [ <union_arm> ]... } <union_name> ;
   }
```

  If an IDL <union_arm> is empty (that is, an instance resolves to only a **;** (semicolon)), then no C mapping is generated for that arm. If <union_name> is not present in the IDL declaration, then the union is assigned the name **tagged_union**.

- The IDL pipe declaration:

```
pipe <type_spec> <pipe_declarators>
```

  maps to the C language as:

```
typedef
   struct <pipe_declarators> {
      void (*pull) (
         char *state,
         <type_spec> *buf,
         idl_ulong_int esize,
         idl_ulong_int *ecount
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        );
        void (*push) (
           char *state,
           <type_spec> *buf,
           idl_ulong_int *ecount
        );
        void (*alloc) (
           char *state,
           idl_ulong_int bsize,
           <type_spec> **buf,
           idl_ulong_int *bcount
        );
        char *state;
    }  <pipe_declarators> ;
```

- The IDL array declarations in which all bounds of all dimensions evaluate to constants:

```
<Identifier> <[> <const_bounds> <]> [ <[> <const_bounds> <]> ] ...
<const_bounds> ::= <const_exp> | <lower> .. <upper>
<lower> ::= <Integer_literal> | <Identifier>
<upper> ::= <Integer_literal> | <Identifier>
```

  map to the C language as:

```
<Identifier> <[> <size> <]> [ <[> <size> <]> ] ...
```

  where <size> is the value obtained by evaluating either <const_exp> or the expression (<upper>-<lower>+1).

- IDL-conformant array declarations in which the size of the first dimension is determined at run time:

```
<Identifier> <[> [ <star_bounds> ] <]> [  <[> <const_bounds> <]> ] ...
<star_bounds> ::= * | <lower> .. *
<const_bounds> ::= <const_exp> | <lower> .. <upper>
<lower> ::= <Integer_literal> | <Identifier>
<upper> ::= <Integer_literal> | <Identifier>
```

  when used in type definitions map to the C language as:

```
<Identifier> <[> 1 <]> [  <[> <size> <]> ] ...
```

  and when used in parameter declarations map to the C language as:

```
<Identifier> <[> <]> [  <[> <size> <]> ] ...
```

  where <size> is value obtained by evaluating either <const_exp> or the expression (<upper>-<lower>+1).

Mappings to languages other than C will be specified as they become standardised.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex G.
(normative)
Portable Character Set

The portable character set specifies a set of characters that conforming implementations must support. Implementation must be able to encode these characters with the IDL **char** data type, but this International Standard does not specify local encodings.

NDR supports two encodings of characters from the portable character set: ASCII and EBCDIC.

Table G-1 on page 560 specifies the NDR encodings in hexadecimal notation.

Note: This table presents only the default conversions. Due to the existence of multiple local variants of EBCDIC, a system manager may provide ASCII/EBCDIC translation tables other than the default.

Characters outside of the Portable Character Set (PCS) are also converted. Special attention should be given to characters outside the PCS because different semantics may be used by applications in different systems. For example, a linefeed character on an EBCDIC system may have different semantics from the linefeed character on an ASCII system since some applications on an ASCII system may take the linefeed character to mean a combination of linefeed and carriage return.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table G-1 Portable Character Set NDR Encodings

Char.	ASCII	EBCDIC	Char.	ASCII	EBCDIC	Char.	ASCII	EBCDIC
(SP)	20	40 21	@	40 4f	7c A	` c1	60 a	79_ 61
"	22	7f	B	42	c2	b	62	82
#	23	7b	C	43	c3	c	63	83
$	24	5b	D	44	c4	d	64	84
%	25	6c	E	45	c5	e	65	85
&	26	50	F	46	c6	f	66	86
'	27	7d	G	47	c7	g	67	87
(	28	4d	H	48	c8	h	68	88
)	29	5d	I	49	c9	i	69	89
*	2a	5c	J	4a	d1	j	6a	91
+	2b	4e	K	4b	d2	k	6b	92
,	2c	6b	L	4c	d3	l	6c	93
-	2d	60	M	4d	d4	m	6d	94
.	2e	4b	N	4e	d5	n	6e	95
/	2f	61	O	4f	d6	o	6f	96
0	30	f0	P	50	d7	p	70	97
1	31	f1	Q	51	d8	q	71	98
2	32	f2	R	52	d9	r	72	99
3	33	f3	S	53	e2	s	73	a2
4	34	f4	T	54	e3	t	74	a3
5	35	f5	U	55	e4	u	75	a4
6	36	f6	V	56	e5	v	76	a5
7	37	f7	W	57	e6	w	77	a6
8	38	f8	X	58	e7	x	78	a7
9	39	f9	Y	59	e8	y	79	a8
:	3a	7a	Z	5a	e9	z	7a	a9
;	3b	5e	[	5b	4a	{	7b	c0
<	3c	4c	\	5c	e0	|	7c	bb
=	3d	7e	]	5d	5a	}	7d	d0
>	3e	6e	^	5e	5f	~	7e	a1
?	3f	6f	_	5f	6d			

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex H.
(normative)
Endpoint Mapper Well-known Ports

The well-known ports used by the endpoint mapper are assigned by the appropriate authority for each protocol. Table H-1 lists the well-known ports currently assigned to the endpoint mapper.

Table H-1 Endpoint Mapper Well-known Ports

Protocol	Port
DOD TCP	135
DOD UDP	135
DECnet Phase IV	69
DECnet Phase V	69
Domain DDS	12

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex I.
(normative)
Protocol Identifiers

This Annex defines the protocol identifiers that are used in protocol towers. Three types of protocol identifiers are supported:

- An octet string derived from OSI Object Identifiers (OIDs). The prefix of this protocol identifier (first octet) has the value 0 (zero). The suffix consists of the OSI OID encoded in ASN.1/BER. Companies can assign their own values by using their OIDs.
- An octet string derived from an interface UUID combined with a version number. This type (**UUID_type_identifier**) can be used for dynamically generated protocol identifiers where registration is not sufficient or desired. The encoding of this protocol identifier type is as follows:
  - — Octet 0 contains the hexadecimal value 0d. This is a reserved protocol identifier prefix that indicates that the protocol ID is UUID derived.
  - — Octets 1 to 16 inclusive contain the UUID, in little-endian format.
  - — Octets 17 to 18 inclusive contain the major version number, in little-endian format.

  OSF maintains a registry of transfer syntax identifiers encoded by using **UUID_type_identifier**s. Currently there is one registered value for NDR. Hexadecimal values for the NDR syntax identifier fields are shown in Table I-1.

Table I-1 NDR Transfer Syntax Identifier

Prefix	UUID	Version	Comments
0d	8a885d04-1ceb-11c9-9fe8-08002b104860	01	Version 1.1 data representation protocol.

Note: Contact OSF to obtain a listing of registered transfer syntaxes, including optional transfer syntaxes not specified by this International Standard, or to register new transfer syntaxes.

- Single octet identifiers that are registered by the Open Software Foundation for commonly used protocols. Table I-2 on page 564 lists currently registered values.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table I-2 Registered Single Octet Protocol Identifiers

Protocol ID for:	Identifier Value	Related Information	Comments
OSI TP4	05	T-Selector	
OSI CLNS	06	NSAP	
DOD TCP	07	port	port address is 16 bit unsigned integer, big endian order
DOD UDP	08	port	port address is 16 bit unsigned integer, big endian_order
DOD IP	09	host address	host address is 4 octets, big endian order
RPC connectionless protocol	0a	minor version	major version 4
RPC connection-oriented protocol	0b	minor version	major version 5
DNA Session Control	02	—	
DNA Session Control V3	03	—	
DNA NSP Transport	04	—	
DNA Routing	06	—	
Named Pipes	10	—	Microsoft Named Pipes
NetBIOS	11	—	Microsoft NetBIOS
NetBEUI	12	—	Microsoft NetBEUI
Netware SPX	13	—	Netware SPX transport-layer protocol
Netware IPX	14	—	Netware IPX transport-layer protocol

Note: Contact OSF to register a protocol identifier or for the format and semantics of the **Related Information** entries not given in the table.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex J.
(normative)
DCE CDS Attribute Names

Table J-1 lists attribute name mapping for OSI object identifiers for DCE CDS.

Table J-1 DCE CDS Attribute Names

Attribute Name	OSI Object Identifier
CDS_Class	1.3.22.1.3.15 {iso(1) identified-org(3) osf(22) dce(1) cds(3) CDS_Class(15)}
CDS_ClassVersion	1.3.22.1.3.16 {iso(1) identified-org(3) osf(22) dce(1) cds(3) CDS_ClassVersion(16)}
RPC_ClassVersion	1.3.22.1.1.1 {iso(1) identified-org(3) osf(22) dce(1) rpc(1) RPC_ClassVersion(1)}
RPC_ObjectUUIDs	1.3.22.1.1.2 {iso(1) identified-org(3) osf(22) dce(1) rpc(1) RPC_ObjectUUIDs(2)}
CDS_Towers	1.3.22.1.3.30 {iso(1) identified-org(3) osf(22) dce(1) cds(3) CDS_Towers(16)}
RPC_Group	1.3.22.1.1.3 {iso(1) identified-org(3) osf(22) dce(1) rpc(1) RPC_Group(3)}
RPC_Profile	1.3.22.1.1.4 {iso(1) identified-org(3) osf(22) dce(1) rpc(1) RPC_Profile(4)}

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex K.
(normative)
Architected and Default Values for Protocol Machines

Table K-1 identifies the recommended default time-out values referenced in the client and server protocol machines. Implementations must provide for default settings of these timers. Applications may overwrite these default values through appropriate API functions.

Table K-1 Default Protocol Machine Values

Reference Name	Protocol	Default Value	Description
TIMEOUT_ACK	CL	3 seconds	Wait before sending an **ack** PDU.
TIMEOUT_BROADCAST	CL	5 seconds	Wait for a response to a **broadcast** PDU.
TIMEOUT_CANCEL	CL	1 second	Wait for a response to a **cancel** PDU.
TIMEOUT_CANCEL	CO	Infinity	Wait for a response to a **cancel** PDU.
TIMEOUT_FRAG	CL	1 second	Wait for a **fack** PDU if the **no_fack** flag was cleared.
TIMEOUT_IDLE	CL	300 seconds	Time for keeping state information about the client.
TIMEOUT_MAX_ALLOC_WAIT	CO	3 seconds	Initial value for wait before retrying association allocation.
TIMEOUT_PING	CL	1 second	Wait for a response to a **ping** PDU.
TIMEOUT_RESEND	CL	3 seconds	Wait for acknowledgement from client before retransmitting a response.
TIMEOUT_SERVER_DISCONNECT	CO	10 seconds	Wait before shutdown of idle connections (only if resources are scarce).
TIMEOUT_WAIT	CL	TBD	Wait for a response to a **request** PDU.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex L.
(normative)
Protocol Tower Encoding

This Annex details the encoding of RPC binding information as protocol towers.

Subclause 6.2.3.1 on page 277 describes an abstract model of RPC binding information referred to as a **protocol_tower_t** data type.

Annex N defines the actual (concrete) IDL representation of a protocol tower data type as **twr_t** and **twr_p_t** data types as follows:

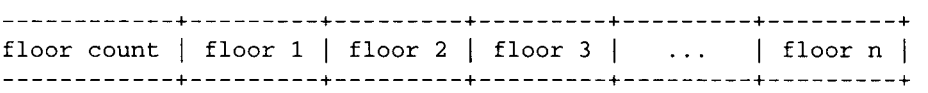
```
/*
 * Protocol Tower. The network representation of network addressing
 * information (e.g., RPC bindings).
 */
typedef struct {
    unsigned32          tower_length;
    [size_is(tower_length)]
    byte                tower_octet_string[];
} twr_t, *twr_p_t;
```

This Annex defines the rules for encoding an **protocol_tower_t** (abstract) into the **twr_t.tower_octet_string** and **twr_p_t->tower_octet_string** fields (concrete). For historical reasons, this cannot be done using the standard NDR encoding rules for marshalling and unmarshalling. A special encoding is required.

Note that the **twr_t** and **twr_p_t** are mashalled as standard IDL data types, encoded in the standard transfer syntax (for example, NDR). As far as IDL and NDR are concerned, **tower_octet_string** is simply an opaque conformant byte array. This subclause only defines how to construct this opaque open array of octets, which contains the actual protocol tower information.

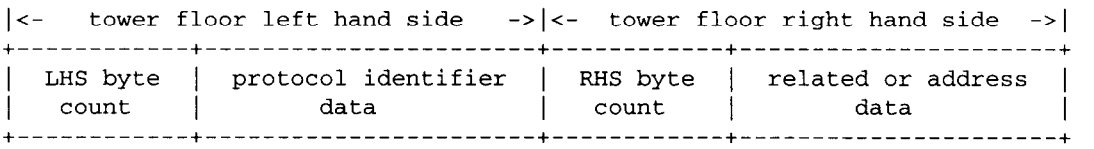
The **tower_octet_string[]** is a variable length array of octets that encodes a single, complete protocol tower. It is encoded as follows:

- Addresses increase, reading from left to right.
- Each **tower_octet_string** begins with a 2-byte floor count, encoded little-endian, followed by the tower floors as follows:



  The number of tower floors is specific to the particular protocol tower, also known as a **protseq**.

- Each tower floor contains the following:



  The **LHS** (Left Hand Side) of the floor contains protocol identifier information. Protocol identifier values and construction rules are defined in Annex I.

  The **RHS** (Right Hand Side) of the floor contains related or addressing information. The type and encoding for the currently defined protocol identifiers are given in Annex I.

  The floor count, **LHS** byte count and **RHS** byte count are all 2-bytes, in little endian format.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

L.1 Protocol Tower Contents

All towers contain the 3 floors shown in Table L-1.

Table L-1 Floors 1 to 3 Inclusive

Floor	Content
1	RPC interface identifier
2	RPC Data representation identifier
3	RPC protocol identifier

The content of floors 4 and 5 are **protseq**-specific. Table L-2 shows the contents for the protocol sequences **ncacn_ip_tcp** and **ncadg_ip_udp**.

Table L-2 Floors 4 and 5 for TCP/IP Protocols

Floor	Content
4	Port address
5	Host address

Implementations may optionally support the protocol sequence **ncacn_dnet_nsp**. Table L-3 shows the tower contents for this protocol.

Table L-3 Floors 4, 5 and 6 for DECnet Protocol

Floor	Content
4	DECnet session control
5	Transport - Network Services Protocol (NSP)
6	DECnet routing - Network Service Access Point (NSAP)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex M.
(normative)
The dce_error_inq_text Manual Page

The **dce_error_inq_text**( ) routine may be used by RPC applications to return message text corresponding to a **status** value. Because this routine is not specifically RPC-related, it is specified in this Annex rather than as part of Section 3.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NAME

dce_error_inq_text — returns the message text for a status code.

SYNOPSIS

```
#include <dce/rpc.h>
#include <dce/dce_error.h>

void dce_error_inq_text(
        unsigned long status_to_convert,
        unsigned char *error_text,
        int *status);
```

ARGUMENTS

Input

status_to_convert Specifies the status code to convert to a text string.

Output

error_text Returns a character string message that corresponds to the **status_to_convert** argument.

status Returns the status code from this operation. A value of 0 (zero) indicates that the operation completed successfully. A value of –1 indicates that it failed.

DESCRIPTION

The **dce_error_inq_text**( ) routine returns a NULL-terminated character string message for the status code specified. The routine uses the value of the environment variable **NLSPATH** to determine the location of the message catalogue from which character string messages are drawn.

The application must provide memory for the returned message. The largest returned message is dce_c_error_string_len characters long, including the terminating NULL character.

If the call fails, this routine returns a message as well as a failure code in the **status** argument.

RETURN VALUE

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex N.
(normative)
IDL Data Type Declarations

This Annex gives IDL type declarations for a variety of data types. Some of these declarations are used only in Annex O, Annex P and Annex Q. If used in an actual implementation, the actual organisation and naming of the IDL import sources including these declarations is implementation-dependent.

N.1 Basic Type Declarations

In this subclause, the interface attribute **[pointer_default(ptr)]** is assumed in effect, unless explicitly overridden.

The following are declarations for integers of specific sizes:

```
typedef unsigned small  unsigned8;
typedef unsigned short  unsigned16;
typedef unsigned long   unsigned32;

typedef small           signed8;
typedef short           signed16;
typedef long            signed32;
```

The following is the declaration for boolean:

```
typedef unsigned32      boolean32;          /* 32-bit wide boolean */
```

The following types are used for status return values:

```
typedef unsigned long   error_status_t;
const long              error_status_ok = 0;
```

The following types are used for UUIDs:

```
typedef struct {
    unsigned32          time_low;
    unsigned16          time_mid;
    unsigned16          time_hi_and_version;
    unsigned8           clock_seq_hi_and_reserved;
    unsigned8           clock_seq_low;
    byte                node[6];
} uuid_t, *uuid_p_t;
```

The following is the declaration for protocol towers, the network representation of network addressing information such as RPC bindings. The contents of the **tower_octet_string** encode the abstract type **protocol_tower_t**, defined in Subclause 6.2.3.1 on page 277 via the encoding rules defined in Annex L, with the **protocol_tower_t** cast into a **byte**[ ] type.

```
typedef struct {
    unsigned32          tower_length;
    [size_is(tower_length)]
    byte                tower_octet_string[];
} twr_t, *twr_p_t;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The following are NDR format flag type definitions and values:

```
const long  ndr_c_int_big_endian    = 0;
const long  ndr_c_int_little_endian = 1;
const long  ndr_c_float_ieee        = 0;
const long  ndr_c_float_vax         = 1;
const long  ndr_c_float_cray        = 2;
const long  ndr_c_float_ibm         = 3;
const long  ndr_c_char_ascii        = 0;
const long  ndr_c_char_ebcdic       = 1;

typedef struct
{
    unsigned8       int_rep;
    unsigned8       char_rep;
    unsigned8       float_rep;
    byte            reserved;
} ndr_format_t, *ndr_format_p_t;
```

The following is the network representation of an IDL context handle:

```
typedef struct ndr_context_handle
{
    unsigned32      context_handle_attributes;
    uuid_t          context_handle_uuid;
} ndr_context_handle;
```

The following are international character types:

```
typedef byte ISO_LATIN_1;

typedef struct
{
    byte            row;
    byte            column;
} ISO_MULTI_LINGUAL;

typedef struct
{
    byte            group;
    byte            plane;
    byte            row;
    byte            column;
} ISO_UCS;
```

The following are authentication protocol IDs. These are architectural values that are carried in RPC protocol messages.

```
const long  dce_c_rpc_authn_protocol_none  = 0;
const long  dce_c_rpc_authn_protocol_krb5  = 1;
typedef unsigned8 dce_rpc_authn_protocol_id_t;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

N.2 Status Codes

This subclause contains declarations for the status codes that may be sent in connectionless **reject** and connectionless and connection-oriented **fault** PDUs. This International Standard specifies the values of these codes; the names are a notational convenience and are not part of the specification.

A distinction can be drawn between protocol-level errors, which are associated with the RPC request/response protocols proper, and application-level errors, which are defined by IDL and the presentation protocol in use (for example, NDR). Errors such as "unknown interface" are in the former category; errors such as "divide-by-zero" are in the latter category.

Protocol-level errors are interpreted by the RPC protocols. They are sent by a server in the body of a connectionless **reject** PDU or in the status field of the header of a connection-oriented **fault** PDU.

Application-level errors are understood by stubs, which map these errors to the appropriate application status return values. In both protocols, application-level errors are indicated in the bodies of **fault** PDUs. The contents of these bodies are untouched by the RPC protocol proper and are simply conveyed from server to client application. In the names below, the application errors are by convention named nca_s_fault_* to distinguish them.

The following status codes are defined:

```
/* unable to get response from server: */
const long  nca_s_comm_failure              = 0x1C010001;
/* bad operation number in call: */
const long  nca_s_op_rng_error              = 0x1C010002;
/* unknown interface: */
const long  nca_s_unk_if                    = 0x1C010003;
/* client passed server wrong server boot time: */
const long  nca_s_wrong_boot_time           = 0x1C010006;
/* a restarted server called back a client: */
const long  nca_s_you_crashed               = 0x1C010009;
/* someone messed up the protocol: */
const long  nca_s_proto_error               = 0x1C01000B;
/* output args too big: */
const long  nca_s_out_args_too_big          = 0x1C010013;
/* server is too busy to handle call: */
const long  nca_s_server_too_busy           = 0x1C010014;
/* string argument longer than declared max len: */
const long  nca_s_fault_string_too_long     = 0x1C010015;
/* no implementation of generic operation for object: */
const long  nca_s_unsupported_type          = 0x1C010017;

const long  nca_s_fault_int_div_by_zero     = 0x1C000001;
const long  nca_s_fault_addr_error          = 0x1C000002;
const long  nca_s_fault_fp_div_zero         = 0x1C000003;
const long  nca_s_fault_fp_underflow        = 0x1C000004;
const long  nca_s_fault_fp_overflow         = 0x1C000005;
const long  nca_s_fault_invalid_tag         = 0x1C000006;
const long  nca_s_fault_invalid_bound       = 0x1C000007;
const long  nca_s_rpc_version_mismatch      = 0x1C000008;
/* call rejected, but no more detail: */
const long  nca_s_unspec_reject             = 0x1C000009;
const long  nca_s_bad_actid                 = 0x1C00000A;
const long  nca_s_who_are_you_failed        = 0x1C00000B;
const long  nca_s_manager_not_entered       = 0x1C00000C;
const long  nca_s_fault_cancel              = 0x1C00000D;
const long  nca_s_fault_ill_inst            = 0x1C00000E;
const long  nca_s_fault_fp_error            = 0x1C00000F;
const long  nca_s_fault_int_overflow        = 0x1C000010;
/* unused:                                    0x1C000011; */
const long  nca_s_fault_unspec              = 0x1C000012;
const long  nca_s_fault_remote_comm_failure = 0x1C000013;
const long  nca_s_fault_pipe_empty          = 0x1C000014;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
const long  nca_s_fault_pipe_closed         = 0x1C000015;
const long  nca_s_fault_pipe_order          = 0x1C000016;
const long  nca_s_fault_pipe_discipline     = 0x1C000017;
const long  nca_s_fault_pipe_comm_error     = 0x1C000018;
const long  nca_s_fault_pipe_memory         = 0x1C000019;
const long  nca_s_fault_context_mismatch    = 0x1C00001A;
const long  nca_s_fault_remote_no_memory    = 0x1C00001B;
const long  nca_s_invalid_pres_context_id   = 0x1C00001C;
const long  nca_s_unsupported_authn_level   = 0x1C00001D;
const long  nca_s_invalid_checksum          = 0x1C00001F;
const long  nca_s_invalid_crc               = 0x1C000020;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

N.3 RPC-specific Data Types

This subclause contains some RPC-specific data types declarations.

In this subclause, the interface attribute **[pointer_default(ref)]** is assumed in effect, unless explicitly overridden.

The following is a declaration of the interface identifier structure, consisting of uuid and major and minor version fields:

```
typedef struct {
    uuid_t                  uuid;
    unsigned16              vers_major;
    unsigned16              vers_minor;
} rpc_if_id_t;
typedef [ptr] rpc_if_id_t *rpc_if_id_p_t;
```

The following is a declaration of a vector of interface identifiers:

```
typedef struct {
    unsigned32              count;
    [size_is(count)]
    rpc_if_id_p_t           if_id[*];
} rpc_if_id_vector_t;
typedef [ptr] rpc_if_id_vector_t *rpc_if_id_vector_p_t;
```

The following are declarations of version options (choices for matching on version numbers):

```
const long  rpc_c_vers_all              = 1;
const long  rpc_c_vers_compatible       = 2;
const long  rpc_c_vers_exact            = 3;
const long  rpc_c_vers_major_only       = 4;
const long  rpc_c_vers_upto             = 5;
```

The following are declarations of constants for accessing values in statistics vector:

```
const long  rpc_c_stats_calls_in            = 0;
const long  rpc_c_stats_calls_out           = 1;
const long  rpc_c_stats_pkts_in             = 2;
const long  rpc_c_stats_pkts_out            = 3;
const long  rpc_c_stats_array_max_size      = 4;
```

The following is a declaration of a statistics vector returned by management inquiry:

```
typedef struct {
    unsigned32          count;
    unsigned32          stats[1];       /* length_is (count) */
} rpc_stats_vector_t, *rpc_stats_vector_p_t;
```

The following are declarations of constants for the endpoint service.

```
const long  rpc_c_ep_all_elts               = 0;
const long  rpc_c_ep_match_by_if            = 1;
const long  rpc_c_ep_match_by_obj           = 2;
const long  rpc_c_ep_match_by_both          = 3;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex O.
(normative)
Endpoint Mapper Interface Definition

This Annex gives the IDL specification of the RPC interface to the endpoint mapper service. It makes use of declarations given in Annex N.

Following are brief descriptions of the operations:

ept_insert()	Add the specified entries to an endpoint map.
ept_delete	Delete the specified entries from an endpoint map.
ept_lookup()	Lookup entries in an endpoint map.
ept_map()	Apply some algorithm (using the fields in the map_tower) to an endpoint map to produce a list of protocol towers.
ept_lookup_handle_free()	Free an ept_lookup or ept_map context_handle.
ept_inq_object()	Inquire Endpoint Map's object id.
ept_mgmt_delete()	Delete matching entries from an endpoint map. All entries that match the tower's interface uuid, version, and network address are deleted. If an object uuid is specified, the entries that are deleted must also match the object uuid.

The endpoint mapper listens on a well-known endpoint for each supported protocol. Registered endpoints are listed in Annex H. An implementation may specify supported endpoints by adding the appropriate endpoint attribute specification to the following declaration.

```
[uuid(e1af8308-5d1f-11c9-91a4-08002b14a0fa), version(3.0),
 pointer_default(ptr)]
interface ept
{
    const long ept_max_annotation_size  = 64;

    typedef struct
    {
                    uuid_t      object;
                    twr_p_t     tower;
        [string]    char        annotation[ept_max_annotation_size];
    } ept_entry_t, *ept_entry_p_t;

    typedef [context_handle] void *ept_lookup_handle_t;

    /*
     * E P T _ I N S E R T
     */

    void ept_insert(
        [in]            handle_t            h,
        [in]            unsigned32          num_ents,
        [in, size_is(num_ents)]
                        ept_entry_t         entries[],
        [in]            boolean32           replace,
        [out]           error_status_t      *status
    );
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
/*
 * E P T _ D E L E T E
 */

void ept_delete(
    [in]            handle_t            h,
    [in]            unsigned32          num_ents,
    [in, size_is(num_ents)]
                    ept_entry_t         entries[],
    [out]           error_status_t      *status
);

/*
 * E P T _ L O O K U P
 */

[idempotent]
void ept_lookup(
    [in]            handle_t            h,
    [in]            unsigned32          inquiry_type,
    [in]            uuid_p_t            object,
    [in]            rpc_if_id_p_t       interface_id,
    [in]            unsigned32          vers_option,
    [in, out]       ept_lookup_handle_t *entry_handle,
    [in]            unsigned32          max_ents,
    [out]           unsigned32          *num_ents,
    [out, length_is(*num_ents), size_is(max_ents)]
                    ept_entry_t         entries[],
    [out]           error_status_t      *status
);

/*
 * E P T _ M A P
 */

[idempotent]
void ept_map(
    [in]            handle_t            h,
    [in]            uuid_p_t            object,
    [in]            twr_p_t             map_tower,
    [in, out]       ept_lookup_handle_t *entry_handle,
    [in]            unsigned32          max_towers,
    [out]           unsigned32          *num_towers,
    [out, length_is(*num_towers), size_is(max_towers)]
                    twr_p_t             towers[],
    [out]           error_status_t      *status
);

/*
 * E P T _ L O O K U P _ H A N D L E _ F R E E
 */

void ept_lookup_handle_free(
    [in]            handle_t            h,
    [in, out]       ept_lookup_handle_t *entry_handle,
    [out]           error_status_t      *status
);

/*
 * E P T _ I N Q _ O B J E C T
 */

[idempotent]
void ept_inq_object(
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        [in]            handle_t            h,
        [out]           uuid_t              *ept_object,
        [out]           error_status_t      *status
    );

    /*
     * E P T _ M G M T _ D E L E T E
     */

    void ept_mgmt_delete(
        [in]            handle_t            h,
        [in]            boolean32           object_speced,
        [in]            uuid_p_t            object,
        [in]            twr_p_t             tower,
        [out]           error_status_t      *status
    );
}
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex P.
(normative)
Conversation Manager Interface Definition

This Annex gives the IDL specification of the conversation manager. It makes use of declarations given in Annex N.

P.1 Server Interface

Following are brief descriptions of the server operations:

conv_who_are_you( )

This operation is called by a server to a client when the server has just received a non-idempotent call request from a client about whom the server knows nothing. The server calls this operation to determine the current sequence number of the client (identified by its activity ID) in question. If the returned sequence number is higher than the one in the request that prompted the **conv_who_are_you**( ) call, the request must be a duplicate and is ignored.

This operation is necessarily idempotent since it supports, and hence can not depend on, non-idempotent call semantics.

It is expected, though not logically required, that servers will maintain a cache of client activity ID/current sequence number pairs to minimise the number of times this operations needs to be called by servers. Cache entries can be dropped as is convenient to servers since the cached information can always be re-obtained by making calls on this operation.

The server passes its boot time back to the client to protect against the case where the server receives a request, executes it, crashes before sending the reply, and then reboots and receives a duplicate of the request. In this scenario, the rebooted server will necessarily make a **conv_who_are_you**( ) call. However, the input boot time will be different and the client, which will have saved the server's boot time from the **conv_who_are_you**( ) call made by the previous incarnation of the server, will notice this and return a non-zero error status to the server, prompting the server to not execute the original request. (Note that the client will still not know whether the call was executed zero or one times. The only guarantee is that it is not executed more than once.)

conv_who_are_you2( )

This is a newer version of **conv_who_are_you**( ) and has a superset of the older call's semantics. The additional semantics are that this call returns a UUID that uniquely identifies the client's address space (CAS UUID). The CAS UUID is used in cases where the server is monitoring the liveness of a client which is not currently making a remote call to the server (for example, in case the server application is holding state on behalf of the client, and it wants to discard or otherwise clean up this state if the client crashes).

For compatibility with old clients (protocol version < 4), servers do not call this operation to get client sequence number information. Rather, they call this operation if, in the course of processing a client's call, they need the CAS UUID. This operation is constructed as a superset of **conv_who_are_you**( ) to make it possible in the future (or in environments with no old clients), for servers to call this operation to get the client's sequence number (and hence avoid making an extra call to get the CAS UUID).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

conv_are_you_there()

Use of **conv_are_you_there**() is implementation-specific (see **convc_indy**() below).

conv_who_are_you_auth()

The version of **conv_who_are_you**() that is used to do authenticated RPC. Instead of calling **conv_who_are_you**() the server can use this function, supplying an authentication challenge to the client, and receive a response to the challenge.

```
[uuid(333a2276-0000-0000-0d00-00809c000000), version(3)]
interface conv
{
    /*
     * C O N V _ W H O _ A R E _ Y O U
     */

    [idempotent]
    void conv_who_are_you(
            [in]    handle_t        h,
            [in]    uuid_t          *actuid,
            [in]    unsigned32      boot_time,
            [out]   unsigned32      *seq,
            [out]   unsigned32      *st
    );

    /*
     * C O N V _ W H O _ A R E _ Y O U 2
     */

    [idempotent]
    void conv_who_are_you2(
            [in]    handle_t        h,
            [in]    uuid_t          *actuid,
            [in]    unsigned32      boot_time,
            [out]   unsigned32      *seq,
            [out]   uuid_t          *cas_uuid,
            [out]   unsigned32      *st
    );

    /*
     * C O N V _ A R E _ Y O U _ T H E R E
     */

    [idempotent]
    void conv_are_you_there(
            [in]    handle_t        h,
            [in]    uuid_t          *actuid,
            [in]    unsigned32      boot_time,
            [out]   unsigned32      *st
    );

    /*
     * C O N V _ W H O _ A R E _ Y O U _ A U T H
     */

    [idempotent]
    void conv_who_are_you_auth(
            [in]    handle_t        h,
            [in]    uuid_t          *actuid,
            [in]    unsigned32      boot_time,
            [in, size_is(in_len)]
                    byte            in_data[],
            [in]    signed32        in_len,
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
            [in]    signed32        out_max_len,
            [out]   unsigned32      *seq,
            [out]   uuid_t          *cas_uuid,
            [out, length_is(*out_len), size_is(out_max_len)]
                    byte            out_data[],
            [out]   signed32        *out_len,
            [out]   unsigned32      *st
    );
}
```

P.2 Client Interface

Following is a brief description of the client operation:

convc_indy()

A client can call this operation to assert its liveness to a server that holds state on its behalf. That is, if a server/client is maintaining liveness and the server does not receive one of these calls within a certain period of time, it will assume the client has died and will notify the server stub routine. Use of **convc_indy**() is implementation-specific, and not specified in this International Standard.

```
[uuid(4a967f14-3000-0000-0d00-012882000000), version(1)]
interface convc
{
    /*
     * C O N V C _ I N D Y
     */

    [maybe]
    void convc_indy(
            [in]    handle_t        h,
            [in]    uuid_t          *cas_uuid
    );
}
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This page intentionally left blank.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex Q.
(normative)
Remote Management Interface

Servers implicitly make available a set of remote management operations which are accessible to applications via **rpc_mgmt_*()** API calls. To support these operations in an interoperable manner, servers must export the remote management interface specified in this Annex. This appendix makes use of data types defined in Annex N.

```
[uuid(afa8bd80-7d8a-11c9-bef4-08002b102989), version(1)]

interface mgmt
{
import "dce/rpctypes.idl";

/*
 * R P C _ _ M G M T _ I N Q _ I F _ I D S
 */

void rpc__mgmt_inq_if_ids
(
    [in]        handle_t                binding_handle,
    [out]       rpc_if_id_vector_p_t    *if_id_vector,
    [out]       error_status_t          *status
);

/*
 * R P C _ _ M G M T _ I N Q _ S T A T S
 */

void rpc__mgmt_inq_stats
(
    [in]        handle_t                binding_handle,
    [in, out]   unsigned32              *count,
    [out, size_is (*count)] unsigned32  statistics[*],
    [out]       error_status_t          *status
);

/*
 * R P C _ _ M G M T _ I S _ S E R V E R _ L I S T E N I N G
 */

boolean32 rpc__mgmt_is_server_listening
(
    [in]        handle_t                binding_handle,
    [out]       error_status_t          *status
);

/*
 * R P C _ _ M G M T _ S T O P _ S E R V E R _ L I S T E N I N G
 */

void rpc__mgmt_stop_server_listening
(
    [in]        handle_t                binding_handle,
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    [out]       error_status_t          *status
);

/*
 * R P C _ _ M G M T _ I N Q _ P R I N C _ N A M E
 */

void rpc__mgmt_inq_princ_name
(
    [in]        handle_t                binding_handle,
    [in]        unsigned32              authn_proto,
    [in]        unsigned32              princ_name_size,
    [out, string, size_is(princ_name_size)]
                char                    princ_name[],
    [out]       error_status_t          *status
);

}
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex R.
(normative)
Mapping of RPC Protocol to an OSI Infrastructure

When the RPC entity is carried by the OSI upper-layers (i.e. RPC is used as an OSI Application Service Element (ASE), the RPC protocol will be treated as a sequence of presentation data values that are carried by the OSI Presentation service, either directly or embedded in other application protocols, (e.g. ISO TP). This annex assigns names to the RPC protocol and the NDR encoding rules that can be used to identify the protocol in OSI Presentation context negotiation.

This annex also assigns a name to the application context consisting of the RPC ASE and the ACSE ASE. Other application context names, including RPC, ACSE and other ASEs may be assigned by other authorities.

R.1 Abstract syntax name

Subclauses 4 and 9 of this International Standard define the RPC abstract syntax, using IDL and the definition, in IDL, of the RPC PDU headers. This International Standard assigns the OSI Object Identifier

{iso(1) standard(0) rpc(11578) abstract-syntax(0) idl(0) }

to the RPC abstract-syntax. This abstract-syntax name can be used to identify the RPC abstract-syntax in the OSI Presentation protocol. The set of presentation data values of the RPC abstract syntax consists of those values, and only those values that are complete RPC pdu's, as defined in clauses 4 and 9 of this International Standard, including all RPC pdu's that contain values specified in a valid IDL specification, as defined in clause 4 of this International Standard.

Note: Other abstract-syntax names may be assigned to abstract syntaxes using the RPC pdus in combination with a particular IDL specification.

R.2 Transfer syntax name

This International Standard assigns the OSI Object Identifier

{iso(1) standard(0) rpc(11578) transfer-syntax(1) ndr(0) }

as a transfer syntax name to the transfer syntax formed by applying the NDR rules specified in clause 14 of this International Standard to an abstract syntax specified in IDL, and in particular, to the RPC abstract syntax. This transfer syntax name shall apply to the use of NDR regardless of the format label and the corresponding encodings of multi-octet values.

Note: Other transfer-syntax names may be assigned to the use of NDR with particular values of the format label and corresponding encodings.

The NDR transfer syntax is octet-aligned and self-delimiting.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

R.3 Simple RPC application context

R.3.1 Application context name

This International Standard assigns the application context name

{iso(1) standard(0) rpc(11578) application-context(2) simple(0) }

to the application context defined in this subclause.

R.3.2 Component ASEs

The component ASEs shall be

the RPC ASE, as specified in the body of this International Standard

the ACSE ASE, as specified in ISO/IEC 8649.
No restrictions are placed on which version of ACSE is used, or which ACSE functional units are used.

Note: The establishment-phase negotiation mechanisms of ACSE will determine which ACSE version and functional units are selected.

R.3.3 Mapping to supporting services

This application context includes both connectionless and connection- oriented modes of RPC and ACSE.

The connectionless RPC pdus shall be mapped to and from the user- information parameter of the A-UNIT-DATA service.

The RPC bind PDU shal be mapped to and from the user information parameter of the A-ASSOCIATE request/indication, or, if issued on an existing association the user data parameter of the Presentation P-DATA service.

The RPC bind_ack and bind_nak PDUs shall be mapped to and from the User Information parameter of the A-ASSOCIATE response service, if the previous RPC bind PDU was received on the User Information parameter of an A-ASSOCIATE indication. Otherwise the RPC bind_ack and bind_nak PDUs shall be mapped to and from the user data parameter of the Presentation P-DATA service.

All other RPC connection-oriented PDUs shall be mapped to and from the user data parameter of the Presentation P-DATA service.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

ICS 35.100.70

Descriptors: data processing, information interchange, network interconnection, open systems interconnection, application layer, data transmission, communication procedure, models.

Price based on 570 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale