

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)



Documents

Document Sharing Hub

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

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

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

	<code>rpc_ns_group_delete()</code>	136
	<code>rpc_ns_group_mbr_add()</code>	137
	<code>rpc_ns_group_mbr_inq_begin()</code>	139
	<code>rpc_ns_group_mbr_inq_done()</code>	141
	<code>rpc_ns_group_mbr_inq_next()</code>	142
	<code>rpc_ns_group_mbr_remove()</code>	144
	<code>rpc_ns_mgmt_binding_unexport()</code>	146
	<code>rpc_ns_mgmt_entry_create()</code>	149
	<code>rpc_ns_mgmt_entry_delete()</code>	150
	<code>rpc_ns_mgmt_entry_inq_if_ids()</code>	151
	<code>rpc_ns_mgmt_handle_set_exp_age()</code>	153
	<code>rpc_ns_mgmt_inq_exp_age()</code>	154
	<code>rpc_ns_mgmt_set_exp_age()</code>	156
	<code>rpc_ns_profile_delete()</code>	158
	<code>rpc_ns_profile_elt_add()</code>	159
	<code>rpc_ns_profile_elt_inq_begin()</code>	161
	<code>rpc_ns_profile_elt_inq_done()</code>	164
	<code>rpc_ns_profile_elt_inq_next()</code>	165
	<code>rpc_ns_profile_elt_remove()</code>	167
	<code>rpc_object_inq_type()</code>	169
	<code>rpc_object_set_inq_fn()</code>	171
	<code>rpc_object_set_type()</code>	172
	<code>rpc_protseq_vector_free()</code>	174
	<code>rpc_server_inq_bindings()</code>	175
	<code>rpc_server_inq_if()</code>	177
	<code>rpc_server_listen()</code>	178
	<code>rpc_server_register_auth_info()</code>	180
	<code>rpc_server_register_if()</code>	182
	<code>rpc_server_unregister_if()</code>	185
	<code>rpc_server_use_all_protseqs()</code>	187
	<code>rpc_server_use_all_protseqs_if()</code>	189
	<code>rpc_server_use_protseq()</code>	191
	<code>rpc_server_use_protseq_ep()</code>	193
	<code>rpc_server_use_protseq_if()</code>	195
	<code>rpc_sm_allocate()</code>	197
	<code>rpc_sm_client_free()</code>	198
	<code>rpc_sm_destroy_client_context()</code>	199
	<code>rpc_sm_enable_allocate()</code>	200
	<code>rpc_sm_free()</code>	201
	<code>rpc_sm_get_thread_handle()</code>	202
	<code>rpc_sm_set_client_alloc_free()</code>	203
	<code>rpc_sm_set_thread_handle()</code>	204
	<code>rpc_sm_swap_client_alloc_free()</code>	205
	<code>rpc_string_binding_compose()</code>	206
	<code>rpc_string_binding_parse()</code>	207
	<code>rpc_string_free()</code>	209
	<code>uuid_compare()</code>	211
	<code>uuid_create()</code>	212
	<code>uuid_create_nil()</code>	213
	<code>uuid_equal()</code>	214
	<code>uuid_from_string()</code>	215
	<code>uuid_is_nil()</code>	216
	<code>uuid_to_string()</code>	217
Section	4	Interface Definition Language
	4.1	Notation
	4.2	IDL Language Specification
	4.2.1	IDL Lexemes
	4.2.1.1	Keywords and Reserved Words
	4.2.1.2	Identifiers

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

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

	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

	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

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

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

	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

	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

	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

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

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.

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.

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.

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.*

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*.

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.

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	Abstract 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

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

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.

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.

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.

- 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.

This page intentionally left blank.

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.

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 invocable 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.

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

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.

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.

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.

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.

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.

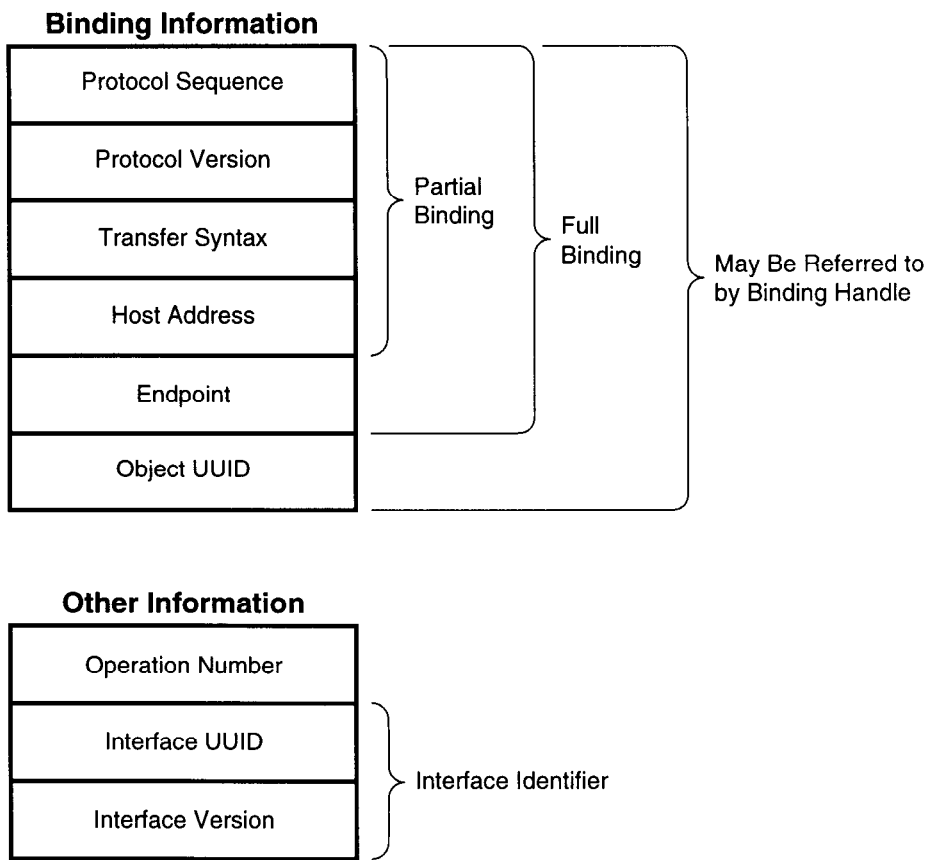


Figure 2-1 Information Required to Complete an RPC

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

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.

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 sub-clauses consider each stage and component in some detail.

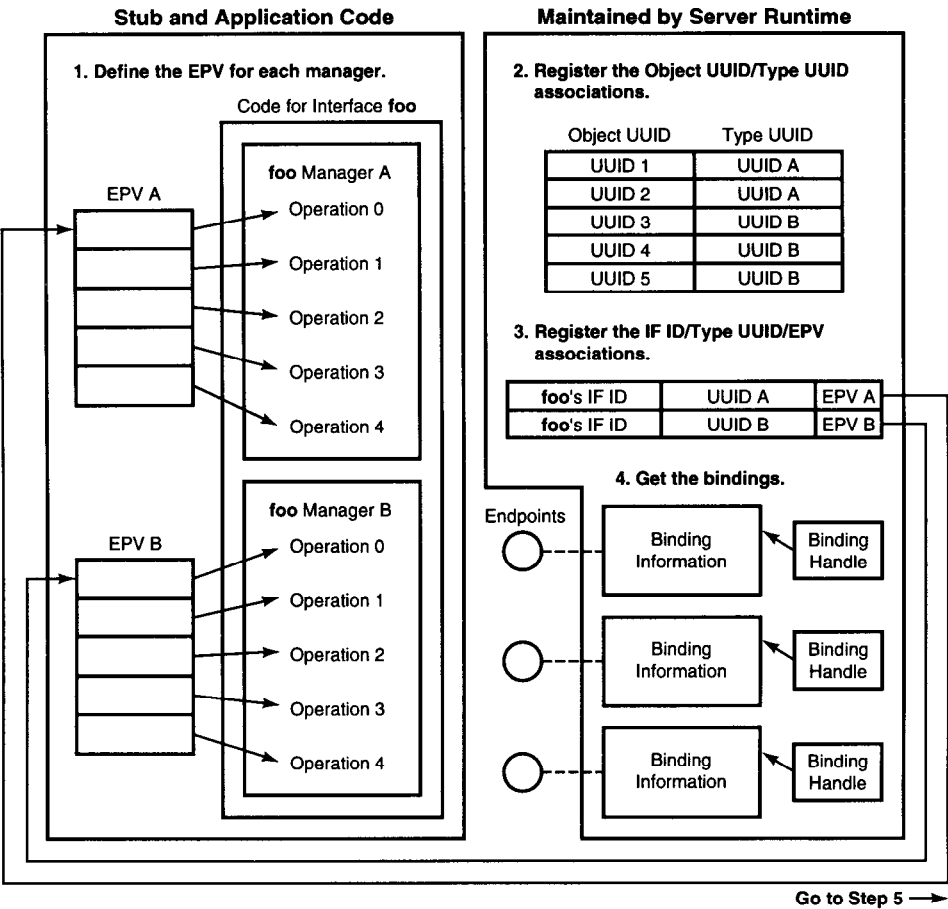
2.3.3.1 Server Binding Steps

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

- by the endpoint mapper
- by a name service.



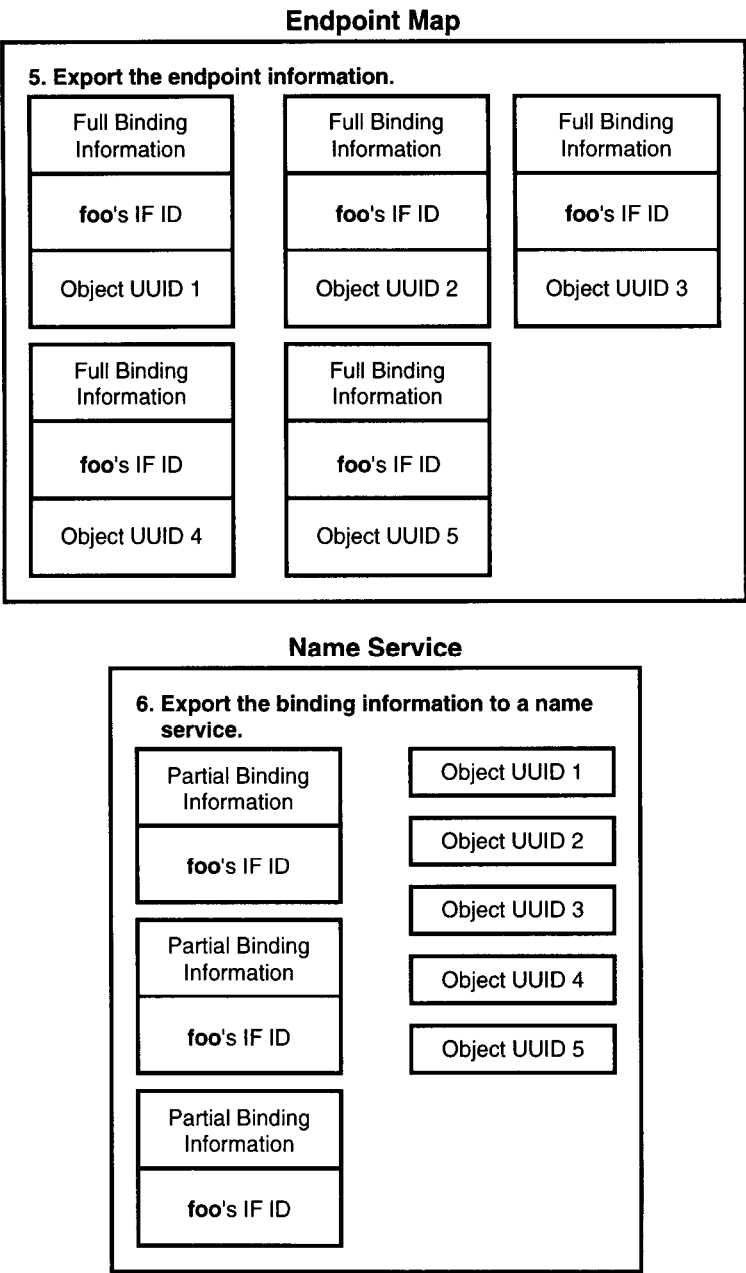


Figure 2-2 Server Binding Relationships

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.

- 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 environment 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.

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.

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.

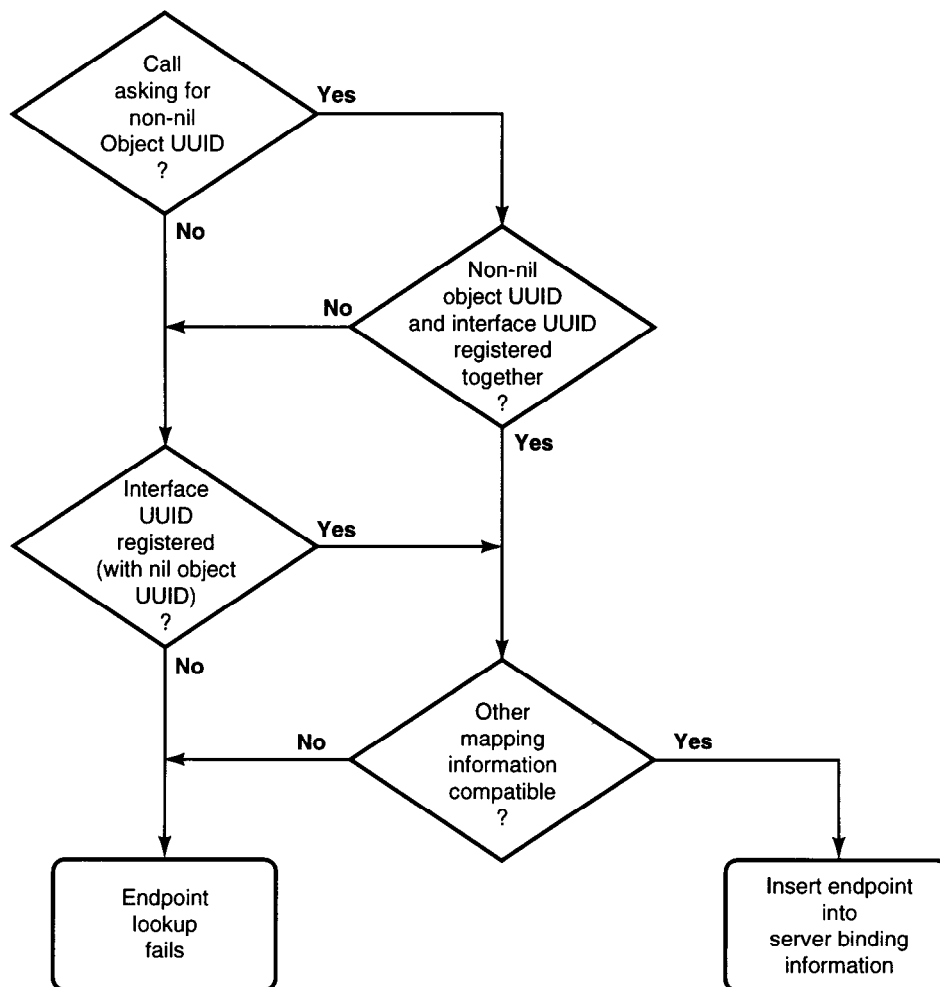


Figure 2-3 Decisions in Looking Up an Endpoint

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

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.

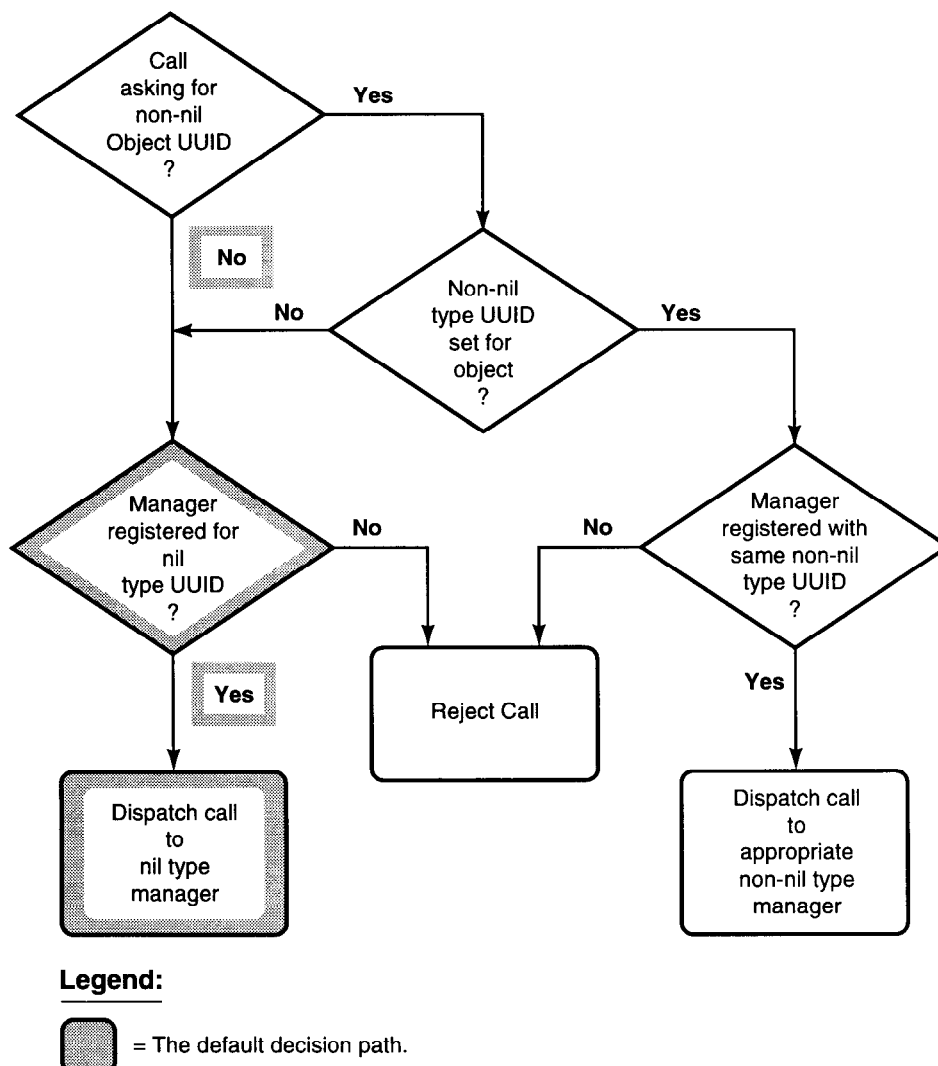


Figure 2-4 Decisions for Selecting a Manager

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.

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.

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.

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.

- 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.

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);
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

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.