

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

ICS 35.100.70

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW



Document Sharing Hub

National foreword

This British Standard reproduces verbatim ISO/IEC 11578:1996 and implements it as the UK national standard.

The UK participation in its preparation was entrusted to Technical Committee IST/21, Open Systems Interconnection, Data Management and Open Distributed Processing, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, pages i and ii, the ISO/IEC title page, pages ii to xviii, pages 1 to 484 and a back cover.

This standard has been updated (see copyright date) and may have had amendments incorporated. This will be indicated in the amendment table on the inside front cover.

This British Standard, having been prepared under the direction of the DISC board, was published under the authority of the Standards Board and comes into effect on 15 June 1997

© BSI 12-1999

ISBN 0 580 27583 3

Amendments issued since publication

Amd. No.	Date	Comments

Contents

	Page
National foreword	Inside front cover
Foreword	xvii
Text of ISO/IEC 11578	1

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)

AVA **Documents**
Document Sharing Hub

Contents

	Page
Foreword	xvii
Section 1. General	
1.1 Scope	1
1.2 Normative references	2
1.3 Informative References	2
1.4 Definitions	2
1.4.1 Reference model definitions	2
1.4.2 Application Layer Structure definitions	2
1.4.3 Service conventions definitions	3
1.5 Abbreviations	3
1.6 Portability	5
1.7 Services and Protocols	5
1.8 Conformance Requirements	6
Section 2. Introduction to the RPC API	
2.1 RPC Programming Model Overview	8
2.1.1 Client/Server Model	8
2.1.1.1 Interfaces	8
2.1.1.2 Remoteness	8
2.1.1.3 Binding	9
2.1.1.4 Name Services	10
2.1.1.5 Resource Models	10
2.1.1.6 Security Services	10
2.1.1.7 Server Implementation	10
2.1.2 Application/Stub/Run-time System	10
2.1.2.1 RPC Run Time	11
2.1.2.2 Stubs	11
2.1.2.3 Application Code	11
2.2 API Operations	11
2.2.1 Binding-related Operations	12
2.2.2 Name Service Operations	12
2.2.3 Endpoint Operations	12
2.2.4 Security Operations	12
2.2.5 Stub Memory Management Operations	12
2.2.6 Management Operations	12
2.2.7 UUID Operations	12
2.3 Binding	12
2.3.1 Binding Handles	14
2.3.1.1 Client and Server Binding Handles	14
2.3.1.2 Obtaining Binding Handles	14
2.3.2 String Bindings	14
2.3.3 Binding Steps	14
2.3.3.1 Server Binding Steps	15
2.3.3.2 Client Binding Steps	18
2.3.3.3 Call Routing Algorithms	18
2.3.4 Binding Methods	21
2.4 Name Service Interface	21
2.4.1 Name Service Model	21
2.4.2 Name Syntax Tags	22
2.4.3 Name Service Attributes	22
2.4.3.1 Server Entries	22

	Page
2.4.3.2 Group Entries	22
2.4.3.3 Profiles	23
2.4.4 Binding Searches	23
2.4.5 Search Algorithm	24
2.4.6 Name Service Caching	25
2.5 Server Model	26
2.5.1 Server Concurrency and Request Buffering	26
2.5.2 Management Interface	26
2.6 Server Resource Models	26
2.6.1 The Server-Oriented Model	26
2.6.2 The Service-Oriented Model	27
2.6.3 The Object-Oriented Model	27
2.7 Security	27
2.8 Error Handling	27
2.9 Cancel Notification	28
2.10 Stubs	28
2.10.1 IDL to Stub Data Type Mappings	28
2.10.2 Manager EPVs	28
2.10.3 Interface Handles	28
2.10.4 Stub Memory Management	29
2.11 RPC API Routine Taxonomy	29
2.11.1 Binding Operations	29
2.11.2 Interface Operations	29
2.11.3 Protocol Sequence Operations	30
2.11.4 Local Endpoint Operations	30
2.11.5 Object Operations	30
2.11.6 Name Service Interface Operations	31
2.11.6.1 NSI Binding Operations	31
2.11.6.2 NSI Entry Operations	31
2.11.6.3 NSI Group Operations	31
2.11.6.4 NSI Profile Operations	31
2.11.7 Authentication Operations	32
2.11.8 The Server Listen Operation	32
2.11.9 The String Free Operation	32
2.11.10 UUID Operations	32
2.11.11 Stub Memory Management	32
2.11.12 Endpoint Management Operations	33
2.11.13 Name Service Management Operations	33
2.11.14 Local Management Services	34
2.11.15 Local/Remote Management Services	34
2.11.16 Error Messages	34
Section 3. RPC API Manual Pages	
3.1 RPC Data Types	35
3.1.1 Unsigned Integer Types	35
3.1.2 Signed Integer Type	35
3.1.3 Unsigned Character String	35
3.1.4 Binding Handle	35
3.1.5 Binding Vector	36

	Page
3.1.6 Boolean Type	37
3.1.7 Endpoint Map Inquiry Handle	37
3.1.8 Interface Handle	37
3.1.9 Interface Identifier	38
3.1.10 Interface Identifier Vector	38
3.1.11 Manager Entry Point Vector	38
3.1.12 Name Service Handle	39
3.1.13 Protocol Sequence String	39
3.1.14 Protocol Sequence Vector	39
3.1.15 Statistics Vector	40
3.1.16 String Binding	40
3.1.17 String UUID	41
3.1.18 UUIDs	41
3.1.19 UUID Vector	41
Section 4. Interface Definition Language	
4.1 Notation	192
4.2 IDL Language Specification	192
4.2.1 IDL Lexemes	192
4.2.1.1 Keywords and Reserved Words	192
4.2.1.2 Identifiers	192
4.2.1.3 IDL Punctuation	193
4.2.1.4 Alternate Representation of Braces	193
4.2.1.5 White Space	193
4.2.2 Comments	193
4.2.3 Interface Definition Structure	193
4.2.4 Interface Header	193
4.2.4.1 The uuid Attribute	194
4.2.4.2 The version Attribute	194
4.2.4.3 The endpoint Attribute	194
4.2.4.4 The local Attribute	194
4.2.4.5 The pointer_default Attribute	194
4.2.5 Interface Body	195
4.2.6 Import Declaration	195
4.2.7 Constant Declaration	195
4.2.7.1 Syntax	195
4.2.7.2 Semantics and Restrictions	196
4.2.8 Type Declarations and Tagged Declarations	197
4.2.9 Base Types	197
4.2.9.1 Syntax	197
4.2.9.2 Integer Types	198
4.2.9.3 The char Types	198
4.2.9.4 The boolean Type	198
4.2.9.5 The byte Type	198
4.2.9.6 The void Type	198
4.2.9.7 The handle_t Type	198
4.2.10 Constructed Types	199
4.2.11 Structures	199
4.2.12 Unions	199

	Page
4.2.12.1 Syntax	199
4.2.12.2 Semantics and Restrictions	200
4.2.13 Enumerated Types	200
4.2.14 Pipes	200
4.2.14.1 Syntax	201
4.2.14.2 Semantics and Restrictions	201
4.2.15 Arrays	201
4.2.15.1 Syntax	201
4.2.15.2 Semantics and Restrictions	201
4.2.15.3 Arrays of Arrays	201
4.2.16 Type Attributes	202
4.2.16.1 Syntax	202
4.2.16.2 Semantics and Restrictions	202
4.2.16.3 The transmit_as Attribute	202
4.2.16.4 The handle Attribute	202
4.2.16.5 The string Attribute	202
4.2.16.6 The context_handle Attribute	203
4.2.17 Field Attributes	203
4.2.17.1 Syntax	203
4.2.17.2 Semantics and Restrictions	203
4.2.17.3 The ignore Attribute	203
4.2.18 Field Attributes in Array Declarations	204
4.2.18.1 Conformant Arrays	204
4.2.18.2 Varying and Conformant Varying Arrays	204
4.2.18.3 Relationships Between Attributes	205
4.2.18.4 Negative Size and Length Specifications	205
4.2.19 Field Attributes in String Declarations	205
4.2.19.1 The first_is, last_is and length_is Attributes	205
4.2.19.2 The max_is Attribute	206
4.2.19.3 The size_is Attribute	206
4.2.20 Pointers	206
4.2.20.1 Syntax	206
4.2.20.2 Semantics and Restrictions	206
4.2.20.3 Attributes Applicable to Pointers	207
4.2.20.4 Varying Arrays of Pointers	207
4.2.20.5 Restrictions on Pointers	208
4.2.21 Pointers as Arrays	208
4.2.21.1 Pointers with the string Attribute	209
4.2.21.2 Possible Ambiguity Resolved	209
4.2.22 Operations	209
4.2.22.1 The idempotent Attribute	209
4.2.22.2 The broadcast Attribute	209
4.2.22.3 The maybe Attribute	210
4.2.23 Parameter Declarations	210
4.2.23.1 Syntax	210
4.2.23.2 Semantics and Restrictions	210
4.2.23.3 Directional Attributes	210
4.2.23.4 Aliasing in Parameter Lists	210

	Page
4.2.24	Predefined Types
4.2.25	The error_status_t Type
4.2.26	International Character Types
4.2.27	Anonymous Types
4.3	The Attribute Configuration Source
4.3.1	Comments
4.3.2	Identifiers
4.3.3	Syntax
4.3.4	Include Declaration
4.3.5	Specifying Binding Handles
4.3.5.1	The explicit_handle Attribute
4.3.5.2	The implicit_handle Attribute
4.3.5.3	The auto_handle Attribute
4.3.6	The represent_as Attribute
4.3.7	The code and nocode Attributes
4.3.8	The in_line and out_of_line Attributes
4.3.9	Return Statuses
4.3.9.1	The comm_status Attribute
4.3.9.2	The fault_status Attribute
4.3.9.3	Interaction of the comm_status and fault_status Attributes
4.3.10	The heap Attribute
4.3.11	The enable_allocate Attribute
4.4	IDL Grammar Synopsis
4.4.1	Grammar Synopsis
4.4.2	Alphabetic Listing of Productions
4.5	IDL Constructed Identifiers
4.6	IDL and ACS Reserved Words
Section 5. Stubs	
5.1	The Application/Stub Interface
5.1.1	Parameters
5.1.1.1	Parameter Memory Management
5.1.1.2	Client-side Allocation
5.1.1.3	Server-side Allocation
5.1.1.4	Aliasing
5.1.2	Default Manager EPVs
5.1.3	Interface Handle
5.1.4	Pipes
5.1.4.1	Processing of in Pipes
5.1.4.2	Processing of out Pipes
5.1.4.3	Processing of in,out Pipes
5.1.5	IDL and ACS Type Attributes
5.1.5.1	The IDL transmit_as Attribute
5.1.5.2	The IDL handle Attribute
5.1.5.3	Interaction of IDL transmit_as and IDL handle Attributes
5.1.5.4	The ACS represent_as Attribute
5.1.5.5	Interaction of the ACS represent_as Attribute and the IDL handle Attribute

	Page
5.1.5.6 Interaction of the ACS represent_as Attribute with the IDL transmit_as Attribute	236
5.1.6 Context Handle Rundown	236
5.2 Interoperability Requirements on Stubs	237
5.2.1 Operation Numbers	237
5.2.2 Error Handling During Floating-Point Unmarshalling	237
Section 6. Remote Procedure Call Model	
6.1 Client/Server Execution Model	239
6.1.1 RPC Interface and RPC Object	239
6.1.1.1 RPC Interfaces	240
6.1.1.2 RPC Objects	240
6.1.2 Interface Version Numbering	240
6.1.2.1 Rules for Changing Version Numbers	240
6.1.2.2 Definition of an Upwardly Compatible Change	241
6.1.2.3 Non-upwardly Compatible Changes	241
6.1.3 Remote Procedure Calls	241
6.1.4 Nested RPCs	241
6.1.5 Execution Semantics	241
6.1.6 Context Handles	242
6.1.7 Threads	242
6.1.8 Cancels	244
6.2 Binding, Addressing and Name Services	245
6.2.1 Binding	245
6.2.2 Endpoints and the Endpoint Mapper	246
6.2.2.1 Client Operation	247
6.2.2.2 Server Operation	247
6.2.3 NSI Interface	247
6.2.3.1 Common Declarations	248
6.2.3.2 Protocol Towers	249
6.2.3.3 The server_name Object Attributes	249
6.2.3.4 The group Object Attributes	250
6.2.3.5 The profile Object Attributes	251
6.2.3.6 Encoding	251
6.2.3.7 Name Service Class Values	251
6.3 Error Handling Model	252
Section 7. RPC Service Definition	
7.1 Call Representation Data Structure	253
7.2 Service Primitives	253
7.2.1 Invoke	253
7.2.2 Result	254
7.2.3 Cancel	254
7.2.4 Error	255
7.2.5 Reject	255
Section 8. Statechart Specification Language Semantics	
8.1 The Elements of Statecharts	257
8.2 State Hierarchies	258
8.3 Concurrency	258
8.4 Graphical Expressions	258

	Page
8.4.1	Default Entrances
8.4.2	Conditional Connectors
8.4.3	Terminal Connectors
8.5	Semantics that Require Special Consideration
8.5.1	Implicit Exits and Entrances (Scope of Transitions)
8.5.2	Conflicting Transitions
8.5.3	Execution Steps and Time
8.5.4	Synchronisation and Race Conditions
8.6	Summary of Language Elements
8.6.1	Event Expressions
8.6.2	Condition Expressions
8.6.3	Action Expressions
8.6.4	Data Item Expressions
8.6.4.1	Atomic Numeric Expressions
8.6.4.2	Compound Numeric Expressions
8.6.4.3	String Expressions
Section 9. RPC Protocol Definitions	
9.1	Conformance
9.2	RPC Stub to Run-time Protocol Machine Interactions
9.2.1	Client Protocol Machines
9.2.2	Server Protocol Machines
9.3	Connection-oriented Protocol
9.3.1	Client/Server
9.3.2	Association Group
9.3.3	Association
9.3.3.1	Association Management Policy
9.3.3.2	Primary and Secondary Endpoint Addresses
9.3.4	Call
9.3.5	Transport Service Requirements
9.4	Connection-oriented Protocol Machines
9.4.1	CO_CLIENT_ALLOC
9.4.2	CO_CLIENT_GROUP
9.4.3	CO_CLIENT
9.4.3.1	ASSOCIATION
9.4.3.2	CONTROL
9.4.3.3	CANCEL
9.4.3.4	CALL
9.4.4	CO_SERVER_GROUP
9.4.5	CO_SERVER
9.4.5.1	ASSOCIATION
9.4.5.2	CONTROL
9.4.5.3	CANCEL
9.4.5.4	WORKING
9.5	Connectionless Protocol
9.5.1	Client/Server
9.5.2	Activity
9.5.3	Call
9.5.4	Maintaining Execution Context and Monitoring Liveness

	Page
9.5.5 Serial Numbers	271
9.5.6 Transport Service Requirements	271
9.6 Connectionless Protocol Machines	271
9.6.1 RPC Stub to Run Time Protocol Machine Interactions	272
9.6.2 CL_CLIENT	272
9.6.2.1 CONTROL	272
9.6.2.2 AUTHENTICATION	272
9.6.2.3 CALLBACK	272
9.6.2.4 PING	272
9.6.2.5 CANCEL	272
9.6.2.6 DATA	272
9.6.3 CL_SERVER	272
9.6.3.1 CONTROL	272
9.6.3.2 AUTHENTICATION	273
9.6.3.3 CANCEL	273
9.6.3.4 WORKING	273
9.7 Naming Conventions	273
Section 10. Connectionless RPC Protocol Machines	
10.1 CL_CLIENT Machine	274
10.1.1 CL_CLIENT Activities	274
10.1.2 CL_CLIENT States	276
10.1.3 CL_CLIENT Events	281
10.1.4 CL_CLIENT Conditions	283
10.1.5 CL_CLIENT Actions	287
10.1.6 CL_CLIENT Data-Items	290
10.2 CL_SERVER Machine	297
10.2.1 CL_SERVER Activities	298
10.2.2 CL_SERVER States	302
10.2.3 CL_SERVER Events	308
10.2.4 CL_SERVER Actions	311
10.2.5 CL_SERVER Conditions	318
10.2.6 CL_SERVER Data-Items	322
Section 11. Connection-oriented RPC Protocol Machines	
11.1 CO_CLIENT Machine	330
11.1.1 CO_CLIENT Activities	330
11.1.2 CO_CLIENT States	332
11.1.3 CO_CLIENT Events	339
11.1.4 CO_CLIENT Actions	343
11.1.5 CO_CLIENT Conditions	349
11.1.6 CO_CLIENT Data-Items	352
11.2 CO_CLIENT_ALLOC Machine	359
11.2.1 CO_CLIENT_ALLOC Activities	359
11.2.2 CO_CLIENT_ALLOC States	359
11.2.3 CO_CLIENT_ALLOC Events	361
11.2.4 CO_CLIENT_ALLOC Actions	363
11.2.5 CO_CLIENT_ALLOC Conditions	363
11.2.6 CO_CLIENT_ALLOC Data-Items	364
11.3 CO_CLIENT_GROUP Machine	364

	Page
11.3.1	CO_CLIENT_GROUP States
11.3.2	CO_CLIENT_GROUP Events
11.3.3	CO_CLIENT_GROUP Data-Items
11.4	CO_SERVER Machine
11.4.1	CO_SERVER Activities
11.4.2	CO_SERVER States
11.4.3	CO_SERVER Events
11.4.4	CO_SERVER Actions
11.4.5	CO_SERVER Conditions
11.4.6	CO_SERVER Data-Items
11.5	CO_SERVER_GROUP Machine
11.5.1	CO_SERVER_GROUP States
11.5.2	CO_SERVER_GROUP Events
11.5.3	CO_SERVER_GROUP Actions
11.5.4	CO_SERVER_GROUP Data-Items
Section 12.	RPC PDU Encodings
12.1	Generic PDU Structure
12.2	Encoding Conventions
12.3	Alignment
12.4	Common Fields
12.4.1	PDU Types
12.4.2	Protocol Version Numbers
12.4.3	Data Representation Format Labels
12.4.4	Reject Status Codes
12.5	Connectionless RPC PDUs
12.5.1	Connectionless PDU Structure
12.5.2	Header Encoding
12.5.2.1	Protocol Version Number
12.5.2.2	PDU Type
12.5.2.3	Flags Fields
12.5.2.4	Data Representation Format Label
12.5.2.5	Serial Number
12.5.2.6	Object Identifier
12.5.2.7	Interface Identifier
12.5.2.8	Activity Identifier
12.5.2.9	Server Boot Time
12.5.2.10	Interface Version
12.5.2.11	Sequence Number
12.5.2.12	Operation Number
12.5.2.13	Interface Hint
12.5.2.14	Activity Hint
12.5.2.15	PDU Body Length
12.5.2.16	Fragment Number
12.5.2.17	Authentication Protocol Identifier
12.5.3	Connectionless PDU Definitions
12.5.3.1	The ack PDU
12.5.3.2	The cancel_ack PDU
12.5.3.3	The cancel PDU

	Page
12.5.3.4 The fack PDU	404
12.5.3.5 The fault PDU	405
12.5.3.6 The nocall PDU	405
12.5.3.7 The ping PDU	405
12.5.3.8 The reject PDU	405
12.5.3.9 The request PDU	406
12.5.3.10 The response PDU	406
12.5.3.11 The working PDU	406
12.6 Connection-oriented RPC PDUs	406
12.6.1 Connection-oriented PDU Structure	407
12.6.2 Fragmentation and Reassembly	407
12.6.3 Connection-oriented PDU Data Types	407
12.6.3.1 Declarations	408
12.6.3.2 Connection-Oriented Protocol Versions	410
12.6.3.3 The frag_length Field	410
12.6.3.4 Context Identifiers	411
12.6.3.5 The call_id Field	411
12.6.3.6 The assoc_group_id Field	411
12.6.3.7 The alloc_hint Field	411
12.6.3.8 Authentication Data	411
12.6.3.9 Optional Connect Reject and Disconnect Data	411
12.6.4 Connection-oriented PDU Definitions	412
12.6.4.1 The alter_context PDU	412
12.6.4.2 The alter_context_resp PDU	413
12.6.4.3 The bind PDU	414
12.6.4.4 The bind_ack PDU	415
12.6.4.5 The bind_nak PDU	416
12.6.4.6 The cancel PDU	416
12.6.4.7 The fault PDU	417
12.6.4.8 The orphaned PDU	418
12.6.4.9 The request PDU	418
12.6.4.10 The response PDU	419
12.6.4.11 The shutdown PDU	420
Section 13. Security	
13.1 The Generic RPC Security Model	421
13.1.1 Generic Operation	421
13.1.2 Generic Encodings	422
13.1.2.1 Protection Levels	422
13.1.2.2 Authentication Services	423
13.1.2.3 Authorisation Services	423
13.1.3 Underlying Security Services Required	423
13.2 Security Services for Connection-oriented Protocol	423
13.2.1 Client Association State Machine	424
13.2.2 Server Association State Machine	424
13.2.3 Sequence Numbers	424
13.2.4 The auth_context_id Field	424
13.2.5 Integrity Protection	425
13.2.6 Connection-oriented Encodings	425

	Page
13.2.6.1 Common Authentication Verifier Encodings	425
13.2.6.2 Encoding for Per-PDU Services	426
13.2.6.3 Credentials Encoding	427
13.3 Security Services for Connectionless Protocol	429
13.3.1 Server Receive Processing	429
13.3.2 Client Receive Processing	429
13.3.3 Conversation Manager Encodings	429
13.3.3.1 Challenge Request Data Encoding	430
13.3.3.2 Response Data Encoding	430
13.3.4 Authentication Verifier Encodings	430
13.3.4.1 dce_c_authn_level_none	431
13.3.4.2 dce_c_authn_level_connect	431
13.3.4.3 dce_c_authn_level_call	431
13.3.4.4 dce_c_authn_level_pkt	431
13.3.4.5 dce_c_authn_level_integrity	431
13.3.4.6 dce_c_authn_level_privacy	431
Section 14. Transfer Syntax NDR	
14.1 Data Representation Format Label	432
14.2 NDR Primitive Types	433
14.2.1 Representation Conventions	433
14.2.2 Alignment of Primitive Types	433
14.2.3 Booleans	434
14.2.4 Characters	434
14.2.5 Integers and Enumerated Types	434
14.2.5.1 Enumerated Types	435
14.2.6 Floating-point Numbers	435
14.2.6.1 IEEE Format	436
14.2.6.2 VAX Format	437
14.2.6.3 Cray Format	438
14.2.6.4 IBM Format	439
14.2.7 Uninterpreted Octets	439
14.3 NDR Constructed Types	440
14.3.1 Representation Conventions	440
14.3.2 Arrays	440
14.3.2.1 Uni-dimensional Fixed Arrays	440
14.3.2.2 Uni-dimensional Conformant Arrays	441
14.3.2.3 Uni-dimensional Varying Arrays	441
14.3.2.4 Uni-dimensional Conformant-varying Arrays	442
14.3.2.5 Ordering of Elements in Multi-dimensional Arrays	442
14.3.2.6 Multi-dimensional Fixed Arrays	442
14.3.2.7 Multi-dimensional Conformant Arrays	442
14.3.2.8 Multi-dimensional Varying Arrays	443
14.3.2.9 Multi-dimensional Conformant and Varying Arrays	443
14.3.3 Strings	444
14.3.3.1 Varying Strings	444
14.3.3.2 Conformant and Varying Strings	444
14.3.4 Arrays of Strings	445
14.3.5 Structures	446

	Page
14.3.5.1 Alignment of Constructed Types	446
14.3.6 Structures Containing Arrays	447
14.3.6.1 Structures Containing a Conformant Array	447
14.3.6.2 Structures Containing a Conformant and Varying Array	448
14.3.7 Unions	448
14.3.8 Pipes	449
14.3.9 Pointers	449
14.3.10 Top-level Pointers	450
14.3.10.1 Top-level Full Pointers	450
14.3.10.2 Top-level Reference Pointers	450
14.3.11 Embedded Pointers	451
14.3.11.1 Embedded Full Pointers	451
14.3.11.2 Embedded Reference Pointers	452
14.3.11.3 Algorithm for Deferral of Referents	452
14.4 NDR Input and Output Streams	452
Annex A (normative) Universal Unique Identifier	454
A.1 Format	454
A.2 Algorithms for Creating a UUID	455
A.2.1 Clock Sequence	455
A.2.2 System Reboot	456
A.2.3 Clock Adjustment	456
A.2.4 Clock Overrun	456
A.2.5 UUID Generation	456
A.3 String Representation of UUIDs	457
A.4 Comparing UUIDs	457
Annex B (normative) Protocol Sequence Strings	458
Annex C (normative) Name Syntax Constants	458
Annex D (normative) Authentication, Authorisation and Protection-level Arguments	459
D.1 The authn_svc Argument	459
D.2 The authz_svc Argument	459
D.3 The protect_level Argument	459
D.4 The privs Argument	460
D.5 The server_princ_name Argument	460
D.6 The auth_identity Argument	460
D.7 Key Functions	460
Annex E (normative) Reject Status Codes and Parameters	462
E.1 Reject Status Codes	462
E.2 Possible Failures	463
E.2.1 comm_status Parameter	463
E.2.2 fault_status Parameter	463
Annex F (normative) IDL to C-language Mappings	464
F.1 Data Type Bindings	464
F.2 Syntax Mappings	466
Annex G (normative) Portable Character Set	467
Annex H (normative) Endpoint Mapper Well-known Ports	469
Annex I (normative) Protocol Identifiers	469
Annex J (normative) DCE CDS Attribute Names	470
Annex K (normative) Architected and Default Values for Protocol Machines	471

	Page
Annex L (normative) Protocol Tower Encoding	471
L.1 Protocol Tower Contents	472
Annex M (normative) The dce_error_inq_text Manual Page	473
Annex N (normative) IDL Data Type Declarations	473
N.1 Basic Type Declarations	473
N.2 Status Codes	475
N.3 RPC-specific Data Types	477
Annex O (normative) Endpoint Mapper Interface Definition	478
Annex P (normative) Conversation Manager Interface Definition	480
P.1 Server Interface	480
P.2 Client Interface	482
Annex Q (normative) Remote Management Interface	482
Annex R (normative) Mapping of RPC Protocol to an OSI Infrastructure	483
R.1 Abstract syntax name	484
R.2 Transfer syntax name	484
R.3 Simple RPC application context	484
R.3.1 Application context name	484
R.3.2 Component ASEs	484
R.3.3 Mapping to supporting services	484
Figure 2-1 — Information Required to Complete an RPC	13
Figure 2-2 — Server Binding Relationships	16
Figure 2-3 — Decisions in Looking Up an Endpoint	19
Figure 2-4 — Decisions for Selecting a Manager	20
Figure 6-1 — Execution Phases of an RPC Thread	243
Figure 6-2 — Concurrent Call Threads Executing in Shared Execution Context	244
Figure 10-1 — CL_CLIENT Statechart	274
Figure 10-2 — CL_SERVER Statechart	297
Figure 11-1 — CO_CLIENT Statechart	330
Figure 11-2 — CO_CLIENT_ALLOC Statechart	359
Figure 11-3 — CO_CLIENT_GROUP Statechart	364
Figure 11-4 — CO_SERVER Statechart	367
Figure 11-5 — CO_SERVER_GROUP Statechart	393
Figure 14-1 — NDR Format Label	432
Figure 14-2 — The Boolean Data Type	434
Figure 14-3 — Character Data Type	434
Figure 14-4 — NDR Integer Formats	435
Figure 14-5 — IEEE Single-precision Floating-point Format	436
Figure 14-6 — IEEE Double-precision Floating-point Format	437
Figure 14-7 — VAX Single-precision (F) Floating-point Format	437
Figure 14-8 — VAX Double-precision (G) Floating-point Format	438
Figure 14-9 — Cray Floating-point Formats	439
Figure 14-10 — IBM Floating-point Formats	439
Figure 14-11 — Uninterpreted Octet Representation	440
Figure 14-12 — Uni-dimensional Fixed Array Representation	441
Figure 14-13 — Uni-dimensional Conformant Array Representation	441
Figure 14-14 — Uni-dimensional Varying Array Representation	441
Figure 14-15 — Uni-dimensional Conformant and Varying Array Representation	442

	Page
Figure 14-16 — Multi-dimensional Fixed Array Representation	442
Figure 14-17 — Multi-dimensional Conformant Array Representation	443
Figure 14-18 — Multi-dimensional Varying Array Representation	443
Figure 14-19 — Multi-dimensional Conformant and Varying Array Representation	444
Figure 14-20 — Varying String Representation	444
Figure 14-21 — Conformant and Varying String Representation	445
Figure 14-22 — Multi-dimensional Conformant and Varying Array of Strings	446
Figure 14-23 — Structure Representation	446
Figure 14-24 — Representation of a Structure Containing a Conformant Array	447
Figure 14-25 — Representation of a Structure Containing a Conformant and Varying Array	448
Figure 14-26 — Union Representation	448
Figure 14-27 — Pipe Representation	449
Figure 14-28 — Top-level Full Pointer Representation	450
Figure 14-29 — Top-level Reference Pointer Representation	451
Figure 14-30 — Embedded Full Pointer Representations	451
Figure 14-31 — Embedded Reference Pointer Representation	452
Figure 14-32 — NDR Input Stream	453
Figure 14-33 — NDR Output Stream	453
Table 3-1 — Client and Server Binding Handles	36
Table 3-2 — Rules for Returning an Object's Type	147
Table 4-1 — Integer Base Types	198
Table 4-2 — IDL Directional Attributes	210
Table 4-3 — Alphabetic Listing of Productions	222
Table 4-4 — Constructed Identifier Classes	225
Table 5-1 — Transmitted Type Routines	234
Table 5-2 — Transferred Type Routines	236
Table 5-3 — Floating Point Error Handling	238
Table 6-1 — Execution Semantics	242
Table 6-2 — Protocol Tower Structure	249
Table 6-3 — The server_name Object Attributes	250
Table 6-4 — RPC-specific Protocol Tower Layers	250
Table 6-5 — Example Protocol Tower	250
Table 6-6 — Service Group Object Attributes	251
Table 6-7 — Configuration Profile Object Attributes	251
Table 7-1 — Invoke Parameters	253
Table 7-2 — Result Parameters	254
Table 7-3 — Cancel Parameters	254
Table 7-4 — Error Parameters	255
Table 7-5 — Reject Parameters	255
Table 8-1 — Events Related to Other Elements	261
Table 8-2 — Compound Events	261
Table 8-3 — Conditions Related to Other Elements	261
Table 8-4 — Compound Conditions	262
Table 8-5 — Actions Related to Other Elements	262

	Page
Table 8-6 — Compound Actions	262
Table 12-1 — RPC Protocol Data Units	397
Table 12-2 — The First Set of PDU Flags	400
Table 12-3 — Second Set of PDU Flags	401
Table 12-4 — Authentication Protocol Identifiers	403
Table 14-1 — NDR Format Label Values	433
Table 14-2 — NDR Floating Point Types	436
Table A-1 — UUID Format	454
Table A-2 — UUID version Field	454
Table A-3 — UUID variant Field	455
Table A-4 — The 4 Msb of clock_seq_hi_and_reserved	456
Table A-5 — Field Order and Type	457
Table B-1 — RPC Protocol Sequence Strings	458
Table C-1 — RPC Name Syntax Defined Constants	458
Table D-1 — Casts for Authorisation Information	460
Table D-2 — RPC Key Acquisition for Authentication Services	461
Table E-1 — Reject Status Codes	462
Table E-2 — Failures Returned in a comm_status Parameter	463
Table E-3 — Failures Returned in a fault_status Parameter	464
Table F-1 — IDL/NDR/C Type Mappings	465
Table F-2 — Recommended Boolean Constant Values	466
Table G-1 — Portable Character Set NDR Encodings	468
Table H-1 — Endpoint Mapper Well-known Ports	469
Table I-1 — NDR Transfer Syntax Identifier	469
Table I-2 — Registered Single Octet Protocol Identifiers	470
Table J-1 — DCE CDS Attribute Names	470
Table K-1 — Default Protocol Machine Values	471
Table L-1 — Floors 1 to 3 Inclusive	472
Table L-2 — Floors 4 and 5 for TCP/IP Protocols	472
Table L-3 — Floors 4, 5 and 6 for DECnet Protocol	472

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*.

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

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 227).

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.

1.7 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 interoperability (see Subclause 5.2 on page 237)
- a specification of information stored in and retrieved from name services (see Subclause 6.2 on page 245, 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).

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

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 18.
- Implementations must support the manager selection rules in **Interface and Manager Selection** on page 19.
- 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 245.
- 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.

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 29 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

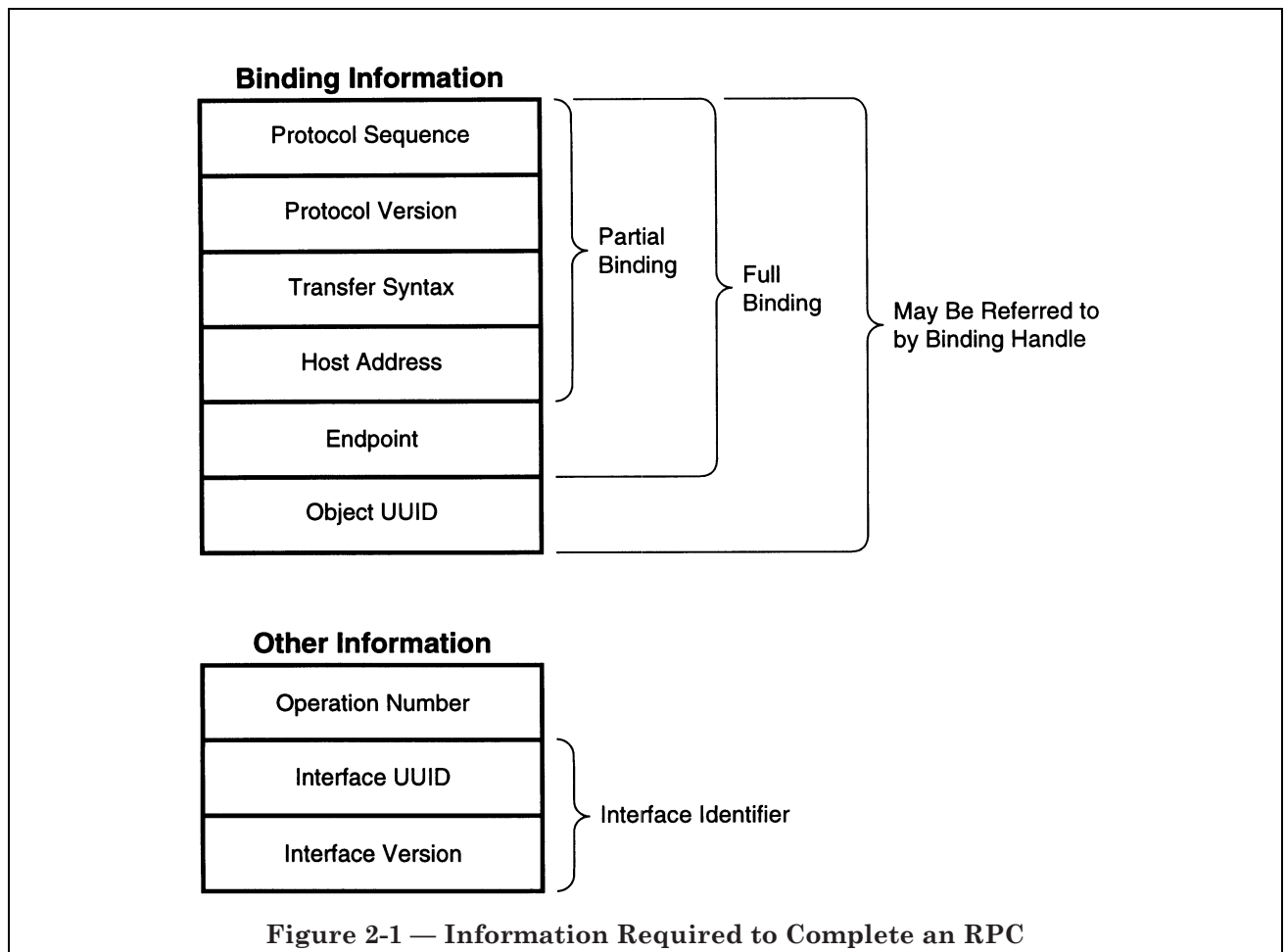
UUIDs (Universal Unique Identifiers) are used frequently by the RPC mechanism for a variety of purposes. The UUID operations enable applications to manipulate UUIDs.

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.



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 13 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 13. 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 19.

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 35 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 35.

2.3.3 Binding Steps

In order to complete an RPC call, all of the elements listed in Figure 2-1 on page 13 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 13 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 15 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

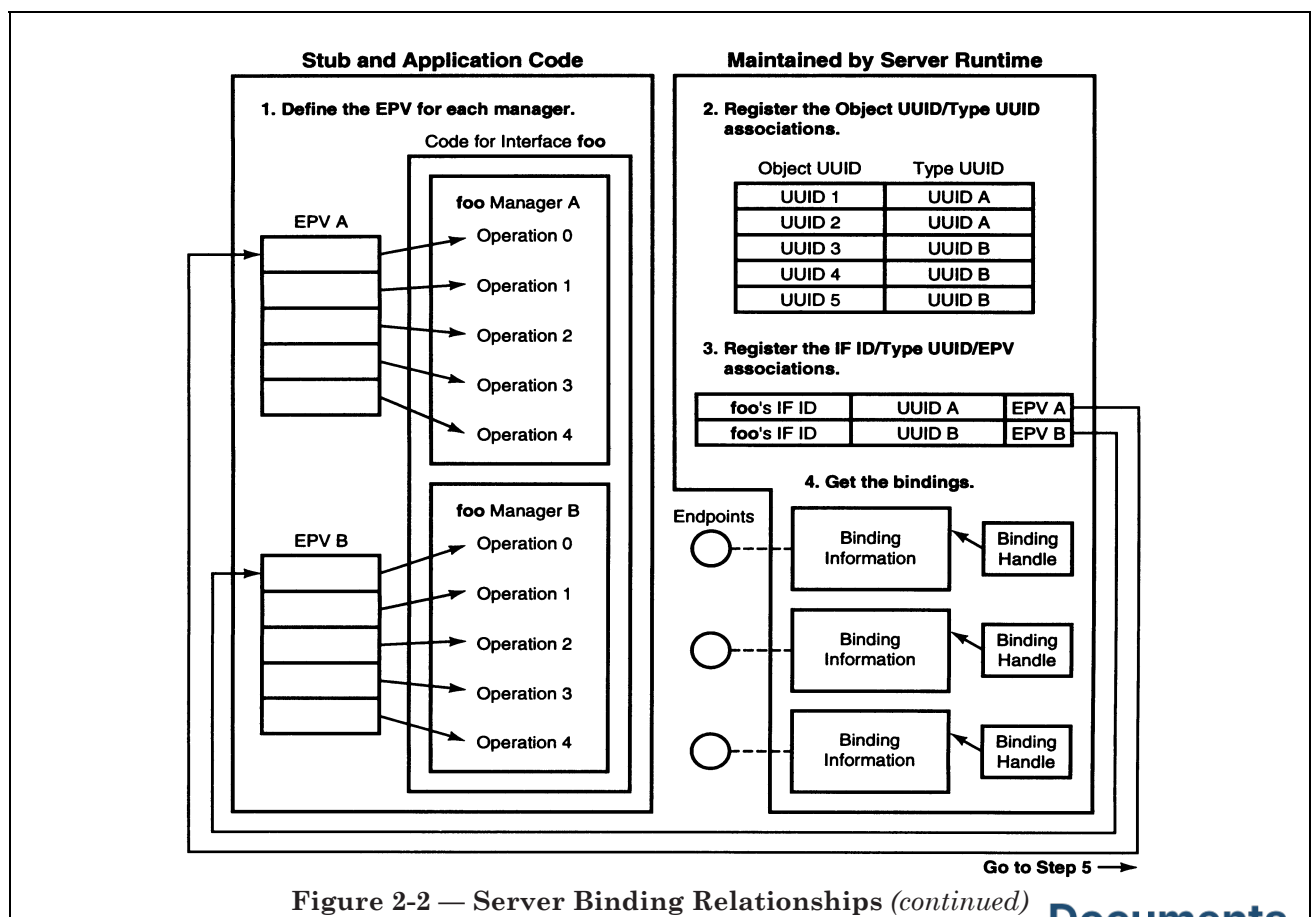


Figure 2-2 — Server Binding Relationships (continued)

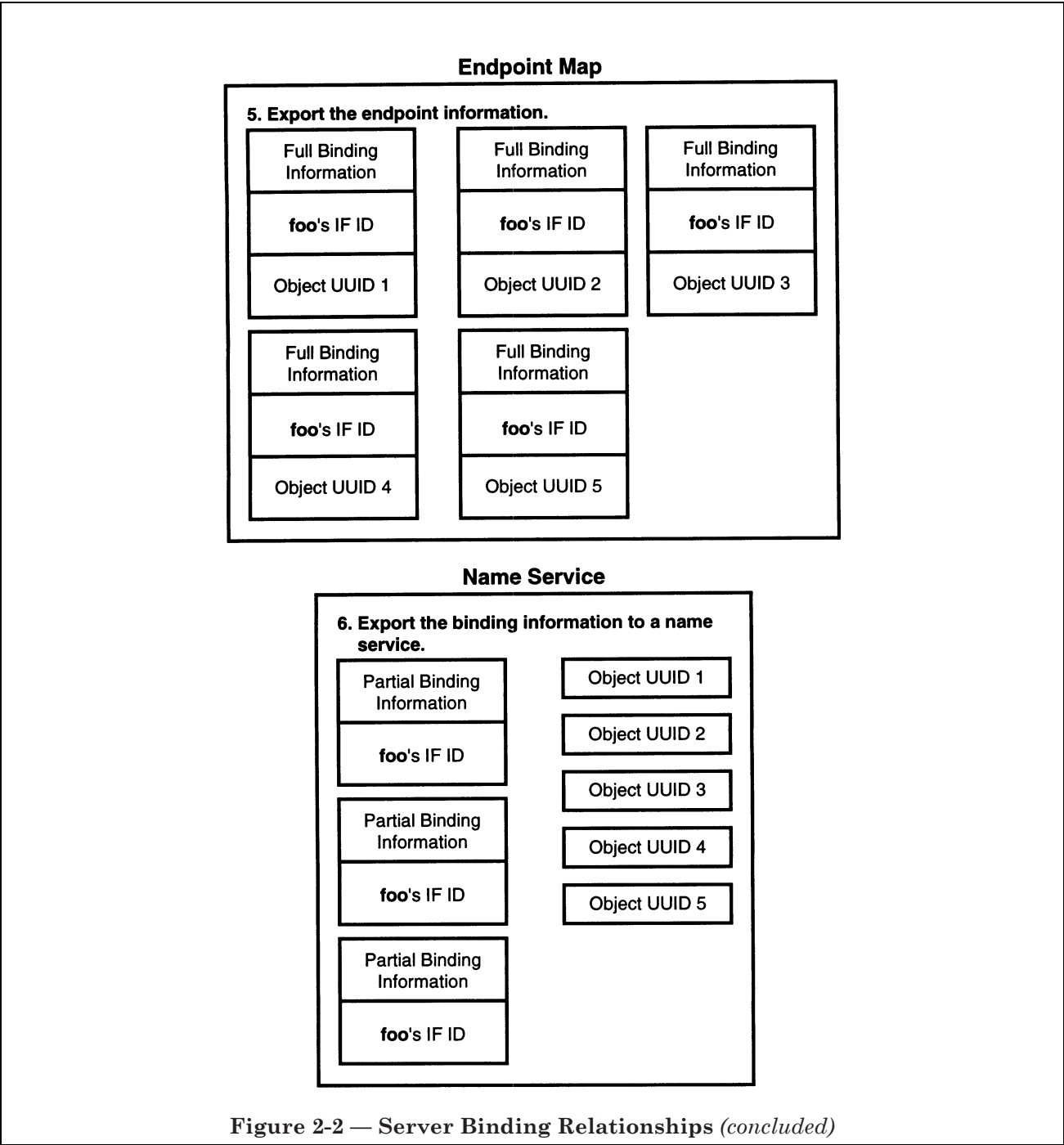


Figure 2-2 — Server Binding Relationships (concluded)

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 21 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 24. 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 26 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 Sub-clause 2.6 on page 26.

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 14 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.

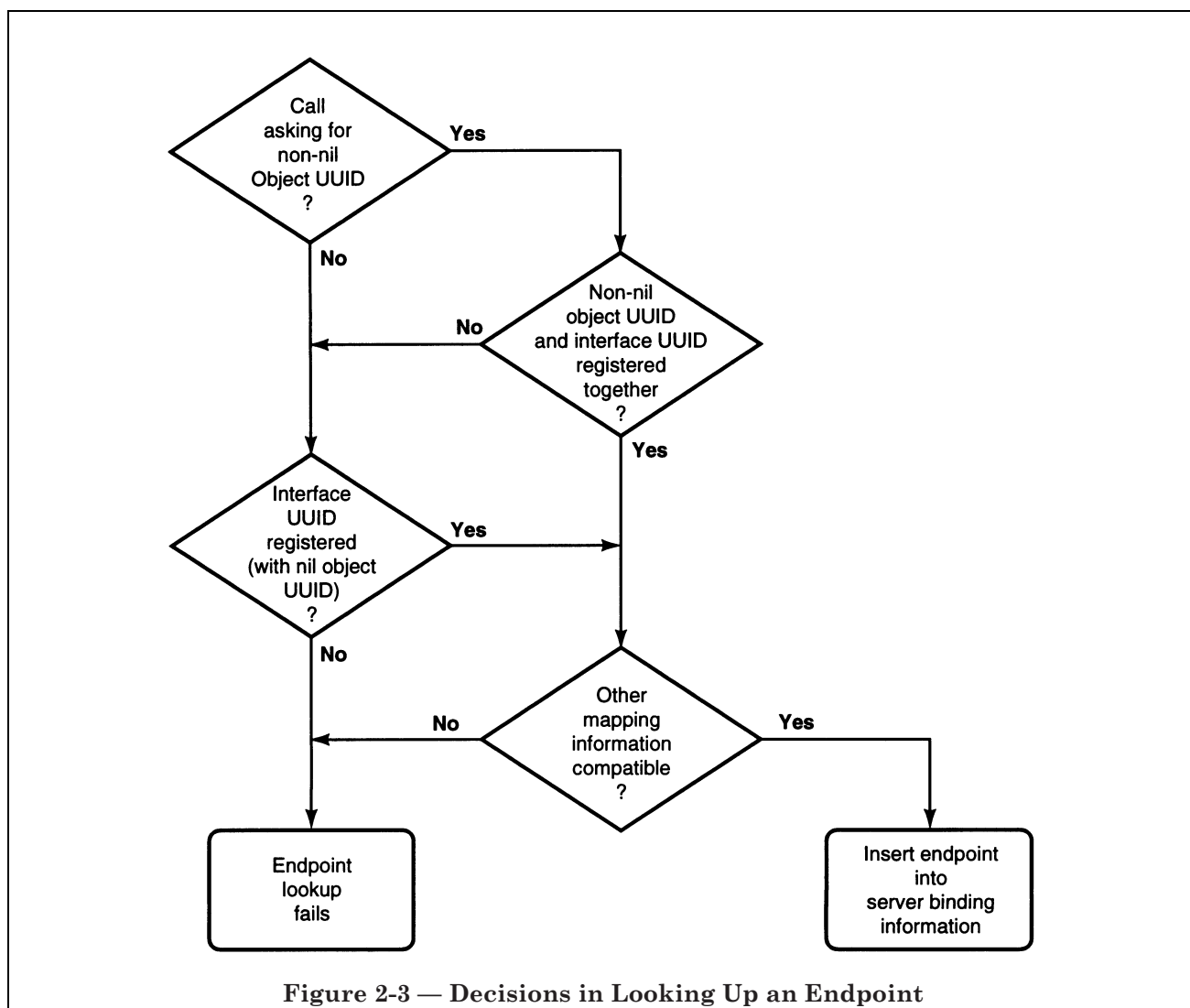
- b) Once the client has found a compatible binding, it makes a call using the binding handle for that binding. Depending on the binding method chosen, the client application code may supply the binding handle explicitly or it may leave this to the stub code (see Subclause 2.3.3 on page 14 and Section 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 16 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.



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.

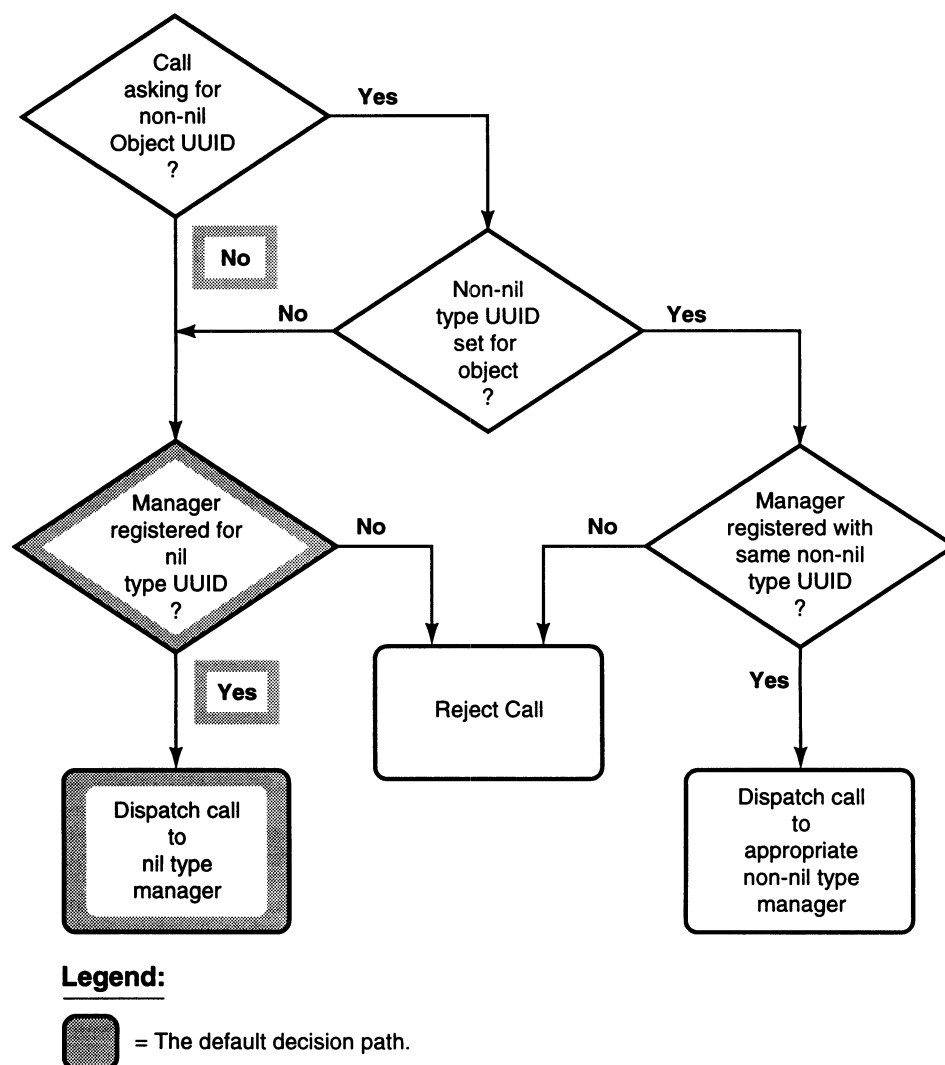


Figure 2-4 — Decisions for Selecting a Manager

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 35 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 18.

- 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 18, 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 **NameService-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 26.

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 22, 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 24. 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 19, 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 26.

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 24 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:

- 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 35.
- 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.
- 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.

2.5 Server Model

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

- support for multiple interfaces, versions, objects and managers, as described in Subclause 2.3 on page 12
- automatic server concurrency and request buffering
- support for remote management.

2.5.1 Server Concurrency and Request Buffering

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

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

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

2.5.2 Management Interface

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

2.6 Server Resource Models

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

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

This International Standard recommends three basic server resource models:

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

These models are not mutually exclusive.

2.6.1 The Server-Oriented Model

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

2.6.2 The Service-Oriented Model

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

2.6.3 The Object-Oriented Model

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

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

Servers that export object UUIDs may or may not explicitly map these to type managers. In the simplest case, the server only registers an interface with a nil type UUID, causing all calls on the interface to be handled by the default manager. In this case, the association between object UUID and resource exists only in the namespace, and the server must assume that a client interested in a given object has, in fact, imported its binding correctly. On the other hand, servers may use object/type mappings to dispatch calls precisely according to object UUID. (See Subclause 2.3.3 on page 14 for the details of the mappings and selection algorithm.)

2.7 Security

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

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

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

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

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

2.8 Error Handling

The RPC API provides a consistent error handling mechanism for all routines. Each routine includes a **status** output argument, which is used to return error status codes. These codes may be passed to the **dce_error_inq_text()** routine to extract error message text from a message catalogue. (See **dce_error_inq_text()** on page 473.)

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

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

2.9 Cancel Notification

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

- a) If the notification is delivered to and handled by the server application thread, the RPC returns normally to the client.
- b) If the server run-time system is unable to deliver the notification to the server application thread (for example, because the server application is blocking notifications), the notification is returned to the client run-time system. The RPC returns normally to the client, and the client run-time system attempts to deliver the notification to the client application thread. The client application code may then handle the notification.
- c) If the notification is delivered to the server application thread, but the server application code fails to handle it, the RPC returns to the client with a fault status.

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

2.10 Stubs

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

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

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

2.10.1 IDL to Stub Data Type Mappings

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

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

2.10.2 Manager EPVs

Stubs may contain a default manager EPV as described in Subclause 3.1 on page 35 Applications that declare additional nondefault manager EPVs must avoid the default name.

2.10.3 Interface Handles

Each stub declares an interface handle, which is a reference to interface specific information that is required by certain RPC APIs. (See Subclause 3.1 on page 35 for an explanation of how applications can access the declared interface handle.)