ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA-127

RPC
BASIC REMOTE PROCEDURE CALL USING OSI REMOTE OPERATIONS

December 1987

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA-127

RPC
BASIC REMOTE PROCEDURE CALL
USING OSI REMOTE OPERATIONS

December 1987

BRIEF HISTORY

Aim

The aim of this ECMA Standard is to facilitate the specification and implementation of Distributed Applications by bringing together programming language techniques, described in Section One, and data-communications techniques, described in Section Two.

Remote Procedure Call (RPC)

Procedures are closed bodies of code which are the units of program modularity in most programming languages. The procedure call is the main construct for executing procedures, and for transferring control and data between them. (See the tutorial in Appendix A.)

Distributed applications consist of separate components in distinct address spaces. Remote Procedure Call (RPC) extends procedure call constructs, so that a procedure in one component of a distributed application can call procedures in other components. The tutorial in Appendix B outlines the concepts and mechanisms involved.

Communications Protocol

A communications protocol is needed to support remote procedure calls. The request/response nature of the Remote Operations (ISO 9072/1 and ISO 9072/2) is ideally suited to the protocol requirements of an RPC in a heterogeneous environment.

This ECMA Standard includes the definitions of ASN.1 (ISO 8824) representations for a set of data types common to ISO Programming Languages. These are used, together with that subset of Remote Operations which is consistent with programming language procedure call semantics, to define the communication protocol for RPC (See Appendix D for further discussion).

Motivations

The main motivations for RPC standardisation are:

i) to enable programmers to use familiar procedure call constructs for remote interactions, avoiding involvement in communications;

ii) to enable the components of distributed applications using RPC to be written in different programming languages and executed in different operating system environments;

iii) to enable distributed applications to be developed in the local environment and become distributed with little or no change;

iv) to enable RPC to be supported by a network conforming to OSI Standards;

v) to specify a standard which is interim to inter-language remote procedure call standardization by ISO.

RPC Standards are an important ingredient in standardization for Open Distributed Processing (ODP).

Adopted as an ECMA Standard by the General Assembly of ECMA on 10th December 1987

TABLE OF CONTENTS

1. GENERAL

1.1 Scope

The subject of this ECMA Standard is Remote Procedure Call (RPC) for open distributed processing, using the ISO 9072/1 and 9072/2 Remote Operations notation, services and protocols. These OSI standards are used without change.

The Standard:

- defines an RPC model;
- defines RPC service primitives to model the procedure invocation constructs common to many (but not all) ISO programming languages;
- defines constructs to support the passing of data types common to many (but not all) ISO programming languages;
- defines RPC error management;
- defines how to specify OSI Application Service Elements with structure appropriate to RPC;
- defines an RPC Application Context;
- defines use of existing OSI protocols to support this RPC Application Context.

The standard does not include specific language bindings for RPC.

The field of application of this standard is: **specifications** of RPC-oriented distributed application interactions, and **implementations** of the protocols to support such applications.

1.2 Conformance

A specification of the remote interactions of a distributed application conforms with this Standard if it satisfies the D1 interface requirement defined in 6.3.

An implementation conforms with this Standard if it implements the protocol requirement specified in 9.1.

Means of conformance testing are not defined in this standard.

No conformance requirements are defined for the internal interfaces of products, or for language bindings. These are matters for future study.

1.3 References

ISO 6093	Representation of Numerical Values in Character Strings for Information Interchange.
ISO 8649	Information Processing Systems - Open Systems Interconnection - Association Control Service Element.
ISO 8824	Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).

ISO 9072/1	Draft International Standard: Information Processing Systems - Text Processing - Remote Operations Part 1: Model, Notation and Service Definition.
ISO 9072/2	Draft International Standard: Information Processing Systems - Text Processing - Remote Operations Part 2: Protocol Specification.
ECMA TR/42	Framework for Distributed Office Applications.

References to documents that are not standards publications are listed in Appendix C.

1.4 Definitions

1.4.1 External Definitions

This Standard uses the following terms defined in other standard documents:

- Application Context:	(ISO 8649)
- Application Entity Title:	(ISO 7498/3)
- Application Service Element:	(ISO 8649)
- Association:	(ISO 8649)
- Association Control Service Element:	(ISO 8649)
- Client:	(ECMA TR/42)
- Remote Operation:	(ISO 9072/1)
- Remote Operation Service Element:	(ISO 9072/1)
- Server:	(ECMA TR/42)
- IMPORT:	(ISO 9072/1)
- EXPORT:	(ISO 9072/2)
- Remote Operation ARGUMENT:	(ISO 9072/1)
- Remote Operation RESULT:	(ISO 9072/1)
- Remote Operation ERRORS:	(ISO 9072/1)
- RO-INVOKE APDU:	(ISO 9072/2)
- RO-RESULT APDU:	(ISO 9072/2)
- RO-ERROR APDU:	(ISO 9072/2)
- RO-REJECT APDU:	(ISO 9072/2).

Note 1

By the above definitions, the terms 'import' and 'export' are matters of macro expansion in ASN.1 module specifications. In this standard these terms are not used with any other meaning.

1.4.2 RPC Definitions

For the purposes of this Standard the following definitions apply. Further definitions are supplied in ASN.1 in the body of the standard.

1.4.2.1 Procedure

A closed sequence of instructions that is entered from and returns control to an external source.

1.4.2.2 Procedure Call

The act of entering a procedure.

1.4.2.3 Procedure Return

The act of returning to the calling procedure.

1.4.2.4 Calling Procedure

A procedure which is the source of a procedure call.

1.4.2.5 Called Procedure

A procedure which is the destination of a procedure call.

1.4.2.6 Remote Procedure Call

A procedure call in which the calling procedure and called procedure do not share address space and may be in physically separate locations.

1.4.2.7 Remote Procedure Call Binding

A particular kind of linkage between a client and a server, which is used for remote procedure calls.

1.4.2.8 Remote Procedure Callback

A remote procedure call in a procedure P, which uses the same RPC binding as the remote procedure call to procedure P.

1.4.2.9 Language Binding

A defined relationship between the syntax and semantics of a programming language and that of some supporting service(s).

1.4.2.10 Input Parameter

An information item which is communicated to the called procedure on entry from the calling procedure.

1.4.2.11 Output Parameter

An information item which is communicated from the called procedure on return to the calling procedure.

1.4.2.12 Input/Output Parameter

An information item which is communicated to the called procedure on entry from the calling procedure, and from the called procedure on return to the calling procedure.

1.4.3 Acronyms

ACSE	association control service element
APDU	application protocol data unit
ASE	application service element
CCR	concurrency, commitment and recovery

PDU	protocol data unit
RO	remote operation
ROSE	remote operation service element
RPC Binding	remote procedure call binding
RPC	remote procedure call

SECTION ONE : RPC STRUCTURE

2. RPC MODEL

2.1 Introduction

A General RPC Model is defined in 2.2. The Open RPC Model defined in 2.3 is a particularisation of it.

2.2 General RPC Model

Functional partitioning for a General RPC Model is defined in this subclause, and is illustrated in Figure 1.

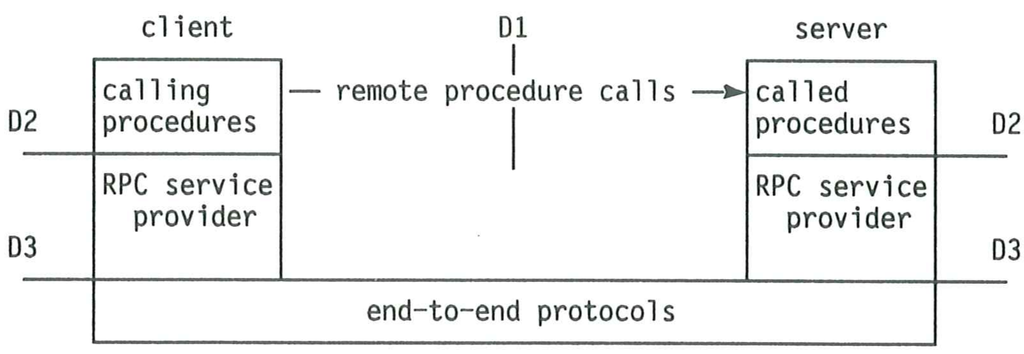


Figure 1 - General RPC Model

This Model may be used to distinguish the main run time components of any RPC, and to position them relative to each other. Three abstract interfaces are identified. They are labelled D1, D2, and D3 (D for distributed).

- **Interface D1** is any **interaction specification** which is implementable via RPC. The content of any particular D1 interface is application-specific, and is defined in an interface declaration. Its generic structure should be constrained to be RPC-like by the specification notation used.
- **Interface D2** is the interface to the **local RPC service** used by application procedures.
- **Interface D3** is the interface to the **end-to-end service** provided by the protocols which mechanise the RPC interactions.

There may also be a level of abstraction at which there is no visible syntactic difference between a local procedure call and a remote procedure call. Such language bindings are outside the scope of this standard.

In the terminology defined in 1.4, the main characteristics of RPC strucuture are as follows. The **remote procedure calls** (and **remote procedure callback**) are made via a **remote procedure call binding** between a **client** and a **server**. These calls invoke execution and pass values between the **calling procedure** the **called procedure** via **input parameters**, **output parmeters** and **input/output parameters**. All parameters are passed by copying values from **client** to **server**, and vice versa.

RPC, viewed as a programming language concept, has a synchronous interaction structure (the "blocking call"), common to local and remote procedure calls. The execution of each procedure call is modelled as a flow of control passing from the calling procedure to the called procedure, and then returning to the calling procedure.

2.3 Open RPC Model

The functional partitioning of Open RPC defined in this standard is illustrated in Figure 2. It is the same as the General RPC Model, but with particular D1, D2 and D3 interfaces.

- **D1**. The D1 remote interactions are defined in the ISO 9072/1 notation (see clause 6) as a named set of Remote Operations, packaged together as an OSI Application Service Element.
- **D2**. The D2 interface is an abstract RPC service (see clause 3).
- **D3**. The D3 interconnection interface is an OSI Application Context defined in the ISO 9072/1 notation (see clause 7).

Items for future study are extensions to these D1, D2 and D3 interfaces, and mappings to other services and end-to-end protocols below D3.

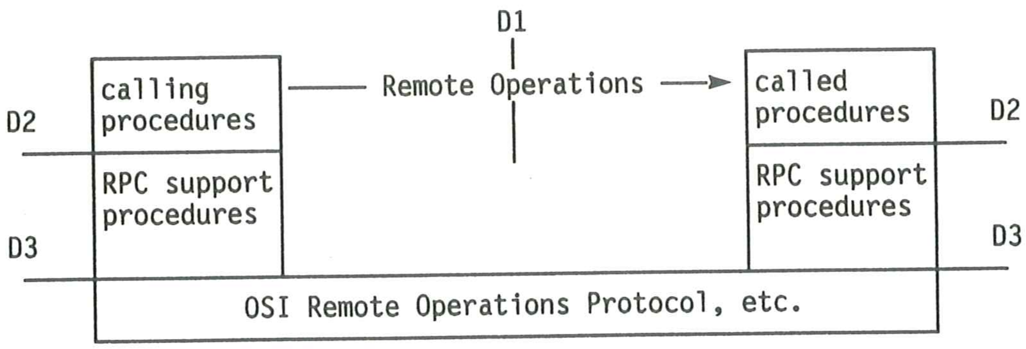


Figure - 2 Open RPC Model

This Standard restricts an RPC Binding to a concurrency level of one (ie. one remote call at a time). Programs with suitable provisions for internal execution concurrency may achieve arbitrary concurrency levels by using multiple separate RPC Bindings.

2.4 Implementation Flexibilities

Some important implementation flexibilities are inherent in this Open RPC Model:

- **Language independence**. The RPC Service and the ISO 9072/1 Remote Operations notation and protocol are independent of choices of programming languages.

- **Machine independence.** The ISO 8824 external data representation used is independent of the host machines.
- **Network independence.** The Open RPC Model is designed to be independent of the choice of protocols below the D3 interface.
- **Implementation independence.** Implementations of this Basic RPC may position the ACSE and ROSE protocol machines as common services, accessed via a communications-oriented read/write interface, or integrate them directly into the RPC support procedures of each program.
- **Hybrid interworking.** Interworking between RPC use of ROSE and non-RPC use of ROSE is also possible, and could facilitate gradual evolution to the RPC style of software construction.

An example of such hybrid interworking is illustrated in Figure 3. This example depends on the application protocol (D1) having been specified in a way which respects the restrictions inherent in the RPC style of interaction. The program at one end implements the interactions at language level as remote procedure calls via the RPC service; but the program at the other end uses a programmed read/write interface to implementations of ACSE and ROSE. Such differences are internal matters. They are masked by the uniformity of: the abstract Remote Operations (D1), the OSI Remote Operations protocol, and the services below the D3 level. These are common to both the implementations.

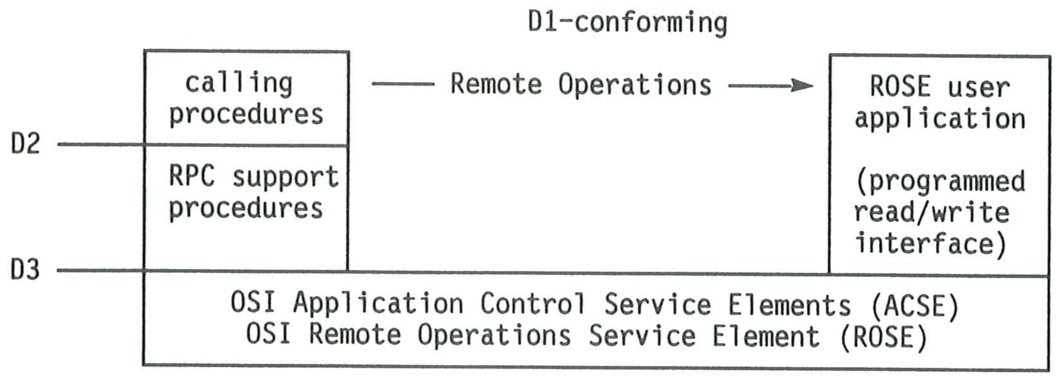

D1-conforming
Remote Operations
calling procedures
RPC support procedures
ROSE user application
(programmed read/write interface)
D2
D3
OSI Application Control Service Elements (ACSE)
OSI Remote Operations Service Element (ROSE)

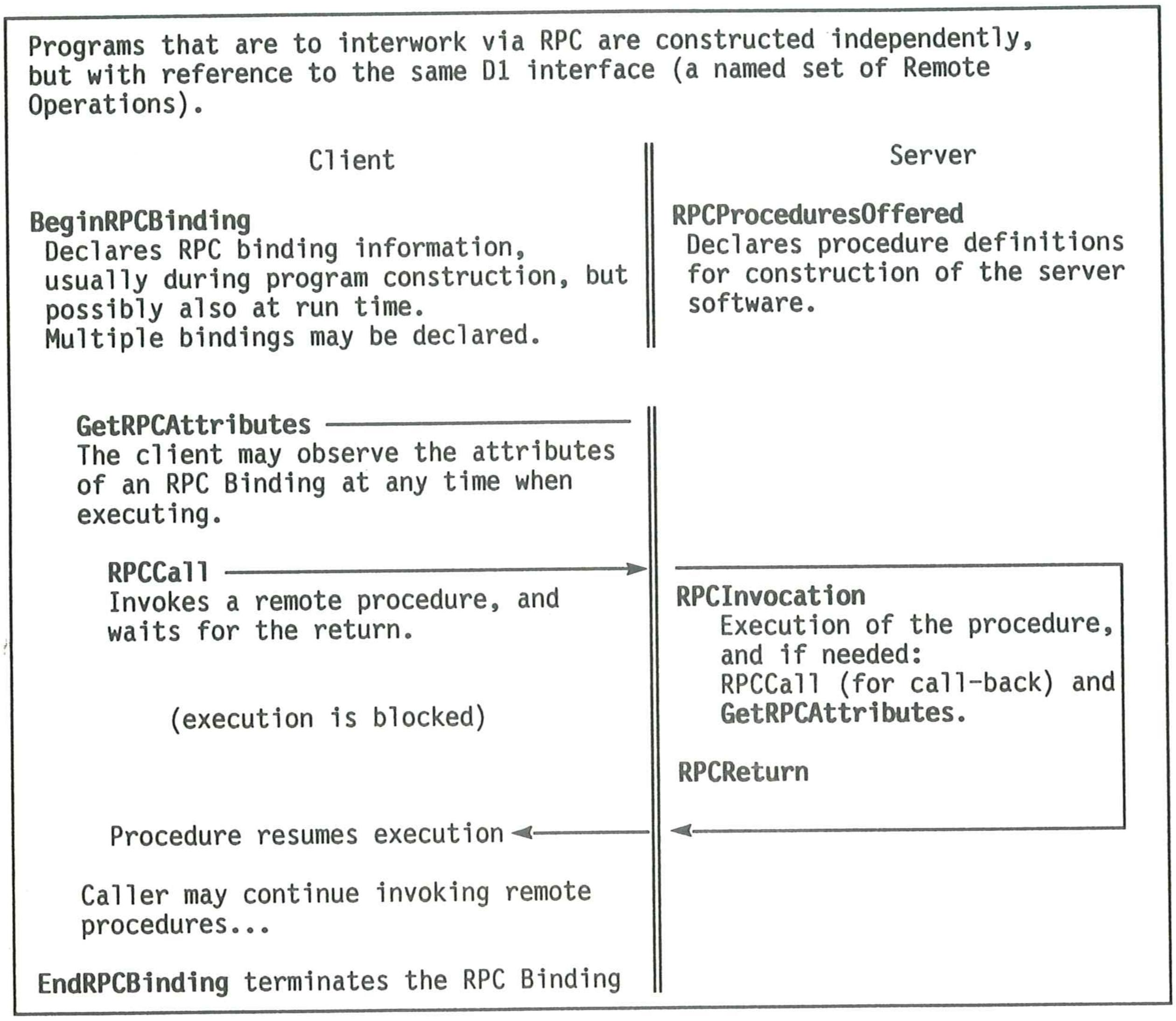
Figure 3 - Hybrid interworking via Remote Operations between an RPC implementation and a communications-oriented implementation

3. RPC SERVICE

3.1 Introduction

The RPC Service at the D2 interface is an abstract model of the programming language constructs used for remote procedure calls.

The service is language-independent. Its main characteristics are illustrated informally in Figure 4 (remote procedure callback is not fully illustrated here).



Programs that are to interwork via RPC are constructed independently, but with reference to the same D1 interface (a named set of Remote Operations).

Client

BeginRPCBinding
Declares RPC binding information, usually during program construction, but possibly also at run time.
Multiple bindings may be declared.

Server

RPCProceduresOffered
Declares procedure definitions for construction of the server software.

GetRPCAttributes
The client may observe the attributes of an RPC Binding at any time when executing.

RPCCall
Invokes a remote procedure, and waits for the return.

(execution is blocked)

RPCInvocation
Execution of the procedure, and if needed:
RPCCall (for call-back) and **GetRPCAttributes.**

RPCReturn

Procedure resumes execution

Caller may continue invoking remote procedures...

EndRPCBinding terminates the RPC Binding

Figure 4 - RPC Service summary

3.2 D2 Interface Specification

The following text through to the end of this subclause is the abstract syntax of the D2 service primitives. It is defined in the ISO 8824 ASN.1 abstract syntax notation. No external data representation or concrete syntax is implied.

```
ECMABasicRPC-D2 DEFINITIONS -- version 1 -- ::=
BEGIN

-- No definitions are exported from this module. It defines the structure and the
-- syntactic content of the service primitives of the Basic RPC D2 interface. This
-- is only defined to the degree necessary for mappings onto D3 (see clause 8),
-- and for integration with RPC parameters and RPC error management (see
-- clauses 4 and 5). The actual language structures corresponding to this
-- interface are implementation-specific.
```

```
--------------------------------------------------------------------------------
-- External definitions:

IMPORTS  RPCBindingStatusInfo, RPCStatusInfo, RPCEnvironmentalError
         FROM ECMABasicRPC-ErrorManagement;

IMPORTS RPCInput, RPCOutput, RPCError, RPCCharacterString
         FROM ECMABasicRPC-CallingSequences
--------------------------------------------------------------------------------
-- First level definitions (ie. the service primitives):

BeginRPCBinding ::= SET {RPCBindingLocalName, RPCBindingClass,
                         ServerName, ServerLocationHint,
                         D1SpecificationReference}
-- Purpose. This primitive is the means for the client to define the beginning of
-- the temporal scope of an RPC Binding, and to propose attribute values. This
-- element does not involve direct interaction between the calling and called
-- procedures. The client may make multiple bindings to the same server or
-- different servers.
--
-- Parameters. The particular RPC Binding is distinguished by the
-- RPCBindingLocalName. The values of the other types are the RPC Binding
-- attributes proposed by the client.
--
-- Implementation Factors. This primitive would be implemented in different ways
-- in different kinds of languages/execution environments. In some it might be a
-- declaration processed at compile time; in others it might be a call to a local
-- support procedure which is executed when the program is loaded; etc. The
-- underlying implementation would, as necessary, access directories, allocate
-- resources, establish communications connections, etc. This would be done
-- before execution of the first GetRPCAttributes or RPCCall element.

EndRPCBinding ::= RPCBindingLocalName
--
-- Purpose. This service primitive is the means for a client program to define the
-- end of the temporal scope of an RPC Binding.
--
-- Parameters. The particular RPC Binding is distinguished by its
-- RPCBindingLocalName value.
--
-- Implementation Factors. This primitive defines the point after which the
-- underlying implementation should release communications resources, etc. In
-- some implementations it might be a declarative statement (E.g. end of block),
-- in others an explicit call to a local support procedure; or it could be implicit in
-- execution termination.
```

```
GetRPCAttributes ::= SET {AttributesReference, RPCBindingStatusInfo,
                          ClientTitle, ClientAddress,
                          ServerTitle, ServerAddress,
                          D1-SpecificationReference, RPCBindingClass}
```

--
-- Purpose. This primitive is the means for client and server to observe the
-- current attributes of an RPC binding.
--
-- Parameters. The D2 user distinguishes the particular RPC Binding by an
-- **AttributesReference** value (which also differentiates between the client and
-- server cases). The D2 service provider is the immediate source of the values
-- of the other types. There is no visibility of any underlying connection identifier
-- or connection endpoint identifier.
--
-- Implementation Factors. This primitive would typically be a procedure call to
-- a local support procedure executed at runtime.

```
RPCCall ::= SET {RemoteProcedureReference, RPCInput,
                 CHOICE {RPCOutput, RPCError, RPCEnvironmentalError}}
```

--
-- Purpose. This primitive is the means for a client or server to invoke execution
-- of a particular remote procedure. A server only uses it for remote procedure
-- callback. Callback may be recursive to arbitrary depth (client calls server, calls
-- back to client, calls back to server...).
--
-- Parameters. The caller provides the **RPCInput** value and the
-- **RemoteProcedureReference** value (which identifies the particular procedure
-- to be executed remotely). The D2 service provider is the direct source of the
-- other values (but their ultimate source is usually the remote procedure that is
-- called).
--
-- Implementation Factors. This primitive would typically be implemented as a
-- procedure call to a local support procedure executed at runtime. This and
-- other support procedures would handle the remote interaction through to its
-- termination; then report the outcome. The usual outcome is **RPCOutput**, but
-- for severe execution errors it is **RPCError**, and for environmental errors it is
-- **RPCEnvironmentalError** (these distinctions are specified in clauses 4 and 5).

```
RPCProceduresOffered ::= SET {SET OF {ProcedureName},
                              D1-SpecificationReference}
```

--
-- Purpose. This primitive models the procedure definitions of the procedures
-- offered for remote execution (including procedures offered for remote
-- procedure callback).
--

```
-- Parameters. The program declares the set of remotely executable procedures
-- corresponding to those identified in the D1 specification which it
-- references.
--
-- Implementation Factors. This primitive would typically be implemented by
-- using a tools package which generates, in the programming language used,
-- the local procedure definitions and procedure headers, etc.

RPCInvocation ::= SET {ProcedureName, RPCInput}
--
-- Purpose. This primitive models invocation of a remotely executable
-- procedure.
--
-- Parameters. The ProcedureName is one of those identified in the
-- RPCProceduresOffered primitive. The RPCInput value comes from the
-- RPCCall primitive.
--
-- Implementation Factors. This primitive would typically be implemented via the
-- normal procedure call construct of the programming language used, coupled
-- with means of dispatching to it the execution of incoming remote requests.

RPCReturn ::= CHOICE {RPCOutput, RPCError}
--
-- Purpose. This primitive models programmed return from a remote
-- procedure.
--
-- Parameters. The exit distinguishes between mutually exclusive outcomes
-- which are defined in clause 4.
--
-- Implementation Factors. This primitive would typically be implemented via the
-- procedure return constructs of the programming language used.

----------------------------------------------------------------------------------------

Second level definitions:

D1-SpecificationReference ::= SEQUENCE
  {specificationName RPCCharacterString,
  specificationIdentifier ObjectIdentifier}
  -- per clause 6.

AttributesReference ::= CHOICE {RPCBindingLocalName, NULL}
-- The client uses an RPCBindingLocalName to reference an RPC Binding via
-- which it calls remote procedures. The NULL reference is for a server to
-- reference the RPC Binding via which it is currently being executed.
```

```
RemoteProcedureReference ::= SET {RPCBindingLocalName, ProcedureName}
-- Together these identify the remote procedure to be executed.
```

Third level definitions:

```
RPCBindingLocalName ::= ANY
-- References the RPC Binding and thereby the server. Its value is only valid
-- within the lexical scope of the client. In an implementation this might be a
-- programmer defined name, a handle value originating from the D2 service
-- provider, etc.

RPCBindingClass ::= INTEGER {
  staticEvaluationOfBinding      (0),
-- The parameter values of the proposed RPC Binding are to be evaluated and
-- frozen at program construction time.

semiLazyEvaluationOfBinding      (1),
-- The parameter values of the proposed RPC Binding are to be evaluated and
-- acted upon at program initialisation time.

fullyLazyEvaluationOfBinding     (2)}
-- The parameter values of the proposed RPC Binding are to be evaluated and
-- acted upon during program execution, and only at the point when the
-- corresponding language element is actually executed.

ServerName ::= RPCCharacterString
-- The external identity of the server, as visible to the client. These naming
-- conventions may be implementation-specific, and are mapped to OSI naming
-- conventions etc. below the D2 interface (see clause 8).

ServerLocationHint ::= CHOICE {Location, NULL}
-- Means for the client to influence location selection where the named server
-- exists at multiple locations. Also useful if the implementation is not supported
-- by run-time access to Directory Services. Otherwise it is null, and an
-- appropriate location is selected by the D2 service provider, if possible.

ClientTitle    ::= AE-Title
ClientAddress  ::= AE-Address

ServerTitle    ::= AE-Title
ServerAddress  ::= AE-Address
```

```
-- (The RPCError, RPCOutput, RPCError and RPCEnvironmentalError
-- definitions are imported.)

ProcedureName ::= RPCCharacterString
-- Per the programming language used locally. This local name maps in some
-- locally agreed way to the corresponding Operation name in the D1
-- specification being used (see clause 8).

------------------------------------------------------------------------------------------------------

-- Fourth level definitions:

Location ::= RPCCharacterString
-- This is the name/address of the location of the server. The naming/addressing
-- conventions here at the D2 service may be implementation-specific, and are
-- mapped to OSI naming and addressing conventions below the D2 interface
-- (see clause 8).

AE-Title ::= ANY
-- Application-Entity Title established via A-ASSOCIATE. For details see
-- ISO 8649.

AE-Address ::= ANY
-- Presentation address established via A-ASSOCIATE. For details see
-- ISO 8649.

END -- end of ECMA Basic RPC D2 definitions.
```

4. RPC PARAMETERS

4.1 Introduction

This clause defines the parameter structure for remote procedure calls executed via the Basic RPC protocol.

In most programming languages the term "parameter" is used as a synonym for the term "argument". In this clause the term "parameter" is used to refer to values that appear in the programming language statement, while the term "argument" refers to the ISO 9072/1 Operation ARGUMENT.

The parameters of an ISO 9072/2 Operation are not fully self- describing, and the RPC mechanism depends on agreement between the calling and called procedures to interpret the Remote Operations in a consistent way, as defined in this clause.

The defined RPC parameter structure is a stereotyped use of ISO 8824 ASN.1 within ISO 9072/1 Remote Operations, and is intended to be consistent with higher level programming language characteristics. E.g. the error management is defined mostly in terms of status parameters rather than distinct exception/error returns, because that is the way most languages work.

4.2 RPC Calling Sequences

The following text through to the end of this subclause defines the abstract syntax of the RPC parameters. It is in the ISO 8824 ASN.1 abstract syntax notation.

```
ECMABasicRPC-CallingSequences DEFINITIONS -- version 1 -- ::=
   BEGIN

   EXPORTS

-- the module defines the types used to define the structure of Remote
-- Procedure Call parameters:

   RPCInput, RPCOutput, RPCError, -- general parameter structure;

   RPCBoolean, RPCBinaryInteger, RPCFloating, RPCComplex,
   RPCCharacterString, RPCBitString, RPCNumericString, RPCProcedureName,
   RPCDateTime; -- individual parameter types;

--------------------------------------------------------------------------------------------

-- External definitions:


   IMPORTS RPCStatusInfo
           FROM ECMABasicRPC-ErrorManagement;          -- see 5.2


   IMPORTS ERROR
           FROM Remote-Operation-Notation              -- ISO 9072
                {joint-iso-ccittremoteOperations(4)notation(0)};

--------------------------------------------------------------------------------------------

-- First level definitions:


   RPCInput  ::= CHOICE {RPCParameterList, empty}
   -- Metatype for the input parameters of remote procedures (including
   -- input/output parameters).


   RPCOutput ::= SEQUENCE {RPCStatusInfo, RPCParameterList OPTIONAL}
   -- Metaype for the output parameters of remote procedures (including
   -- input/output parameters).


   RPCError  ::= ERROR -- the Remote Operations error return macro.
                 PARAMETER RPCStatusInfo
                 ::= -1
   -- The error return type generic to all remote procedures.
```

```
---------------------------------------------------------------------------------------------

-- Second level definitions:

  RPCParameterList ::= SEQUENCE OF RPCParameter

-- (The definition of RPCStatusInfo is imported.)

---------------------------------------------------------------------------------------------

-- Third level definitions:

  RPCParameter ::= CHOICE {RPCParameterValue,
                           RPCParameterDescriptor}

-- The RPCParameterValue is used for scalar data values except certain types
-- of character strings.
--
-- The RPCParameterDescriptor is used for arrays, records and certain types
-- of strings.
--
-- Comments in D1 interface specifications shall describe the function of each
-- parameter, and whether it is an input parameter, an input/output parameter, or
-- an output parameter.
--
-- Each element in the parameter lists of a D1 Operation shall be one of these
-- types. But the NULL type shall be substituted where an element (parameter)
-- is omitted in a procedure call or procedure return. Parameters should be
-- given meaningful names.

---------------------------------------------------------------------------------------------

-- Fourth level definitions: Parameter Data Types:

  RPCParameterValue ::= CHOICE {
                  RPCBoolean, RPCBinaryInteger,
                  RPCFloating, RPCComplex,
                  RPCCharacterString, RPCBitString, RPCNumericString,
                  RPCProcedureName, RPCDateTime}

---------------------------------------------------------------------------------------------

-- Fifth level definitions: Parameter Data Types:

  RPCBoolean          ::= BOOLEAN
-- ISO 8824 Boolean Type.

  RPCBinaryInteger    ::= INTEGER
-- ISO 8824 Integer Type.
```

```
  RPCFloating        ::= REAL
```
-- This would be the Real Type under consideration for the first Addendum to
-- ISO 8824/8825.

```
  RPCComplex         ::= [APPLICATION 100] IMPLICIT
                         SEQUENCE {realPart REAL, imaginaryPart REAL}
```
-- Floating complex: an ordered pair of quantities, representing a complex
-- number.

```
  RPCCharacterString ::= ISO646String
```
-- character string. The length component of the ISO 8825 encoding
-- is the current length of the string. For fixed length strings,
-- **RPCCharacterString** is an **RPCParameterValue**. For varying length strings,
-- **RPCCharacterString** is contained in an **RPCVaryingStringDescriptor** or an
-- **RPCStringWithBoundsDescriptor**.

```
  RPCBitString       ::= BITSTRING
```
-- ISO 8824 bit string, used here for fixed length bit strings.

```
  RPCNumericString   ::= [APPLICATION 101] IMPLICIT PrintableString
```
-- Numeric string. The values shall conform to ISO 6093. There may also be
-- application-specific restrictions on the values passed.

```
  RPCProcedureName   ::= [APPLICATION 102] IMPLICIT INTEGER
```
-- Integer corresponding to the name of a procedure in the caller's environment
-- which is to be called by a remote procedure callback. The value of
-- **RPCProcedureName** selects one of the Operation values of the LINKED
-- Operations of the Operation (procedure) of which it is a parameter (see the
-- D1 interface, clause 6).

```
  RPCDateTime ::= UTCTime
```
-- ISO 8824 date and time.

-- Fourth level definitions: Parameter Descriptors:

```
RPCParameterDescriptor ::= CHOICE {
   RPCArrayDescriptor, RPCRecordDescriptor,
   RPCVaryingStringDescriptor, RPCStringWithBoundsDescriptor}
```
-- These descriptors are used to describe values that consist of more than one
-- scalar value, or to describe values which otherwise would not be sufficiently
-- described.
--

-- If there is an **RPCArrayDescriptor**, or **RPCVaryingStringDescriptor**, or
-- **RPCStringWithBoundsDescriptor** in the output parameters of a procedure
-- call, there is a corresponding **RPCParameterDescriptor** in the input
-- parameters. It dynamically defines details of the structure acceptable to the
-- calling procedure. This corresponds to the way in which most programming
-- languages communicate such structure via input/output parameters.

-- Fifth level definitions: RPCArrayDescriptor.

```
RPCArrayDescriptor ::= [APPLICATION 103] IMPLICIT
   SEQUENCE {dimensionCount INTEGER,
   -- number of dimensions.

   byColumn BOOLEAN,
   -- If true, the elements of the array are stored by columns (FORTRAN).
   -- That is, the leftmost subscript (first dimension) is varied most rapidly, and
   -- the rightmost subscript (n-th dimension) is varied least rapidly.
   --
   -- If false, the elements of the array are stored by rows (most languages
   -- other than FORTRAN). That is, the rightmost subscript is varied most
   -- rapidly, and the leftmost subscript is varied least rapidly.

RPCArrayBounds,
-- Lower and upper bound for each dimension of the array.

RPCArray OPTIONAL}
-- The values in the array. In RPCInput this optional value is omitted in
-- descriptors for output parameters.

   RPCArrayBounds ::= [APPLICATION 104] IMPLICIT
      SEQUENCE OF RPCBoundForDimension
   -- Defines the set of all lower and upper bounds. Bounds for the i-th
   -- dimension appear as the i-th element of the sequence.

      RPCBoundForDimension ::= [APPLICATION 105] IMPLICIT
                                  SEQUENCE
                                  {RPCLowerBound OPTIONAL,
                                   RPCUpperBound}
      -- When the lower bound for a particular dimension is omitted, its value is
      -- assumed to be 1.

         RPCLowerBound ::= INTEGER
         RPCUpperBound ::= INTEGER
```

```
RPCArray ::= [APPLICATION 106] IMPLICIT
                 SEQUENCE OF {CHOICE
```
-- Elements of the array, presented in the order specified in **byColumn**. Each
-- element shall be of the same type.

```
     {RPCParameterValue,
```
-- Array with scalar elements.

```
     RPCParameterDescriptor}}
```
-- Used to describe an array of records, or varying strings, or strings
-- with bounds. The **RPCParameterDescriptor** shall not be an
-- **RPCArrayDescriptor**.

-- Fifth level definitions: Record/structure Descriptor.

```
RPCRecordDescriptor :: = [APPLICATION 107] IMPLICIT
                               SEQUENCE OF RPCRecordElement
```
-- This descriptor is used to indicate that a parameter is a composite of
-- different **RPCParameterValue** and **RPCParameterDescriptor** types. The
-- **RPCRecordDescriptor** packages the record as a composite whole. It does
-- not define the internal structure (which is defined by each element in the
-- record being an ASN.1 value).

-- An **RPCRecordDescriptor** is not used in RPC Input to describe output
-- parameters.

```
RPCRecordElement ::= RPCParameter
```

-- Fifth level definitions: Varying String Descriptor.

```
RPCVaryingStringDescriptor :: = [APPLICATION 108] IMPLICIT
                                     SEQUENCE { rpcMaxStringLength INTEGER
```
-- Maximum number of characters.

```
RPCCharacterString OPTIONAL}
```
-- Value of the current character string. In **RPCInput** this optional value is
-- omitted in descriptors for output parameters.

```
--------------------------------------------------------------------------------
-- Fifth level definitions: String with Bounds Descriptor.

   RPCStringWithBoundsDescriptor :: = [APPLICATION 109] IMPLICIT
      SEQUENCE {
      RPCBoundForDimension,
   -- Lower and upper bound for the 1-dimension array containing the character
   -- string.

   RPCCharacterString OPTIONAL}
   -- Current value of the string. In RPCInput this optional value is omitted in
   -- descriptors for output parameters.

END -- ECMA Basic RPC Calling Sequences definitions.
```

5. RPC ERROR MANAGEMENT

5.1 Introduction

This clause identifies the different sources of errors which threaten correct operation of remote procedure calls, and defines how they are handled. The term "error" is used here with this general meaning, and does not necessarily imply occurrence of a Remote Operation ERRORS return (RO-ERROR APDU).

A tutorial introduction to error management and related issues is given in A.6, B.3.3 and B.4.2.

5.2 Error Management Specification

The following text through to the end of this subclause is the abstract syntax of the status and diagnostic information via which the Basic RPC service (D2 interface) reports error conditions. The ISO 8824 ASN.1 notation is used.

```
ECMABasicRPC-ErrorManagement DEFINITIONS -- version 1 -- :: =
BEGIN

EXPORTS
-- This module defines:
   RPCBindingStatusInfo, RPCStatusInfo, RPCEnvironmentalError;

--------------------------------------------------------------------------------

-- External definitions:


IMPORTS RPCCharacterString
        FROM ECMABasicRPC-CallingSequences;         -- see 4.2
```

```
----------------------------------------------------------------------------------------------------
-- First level definitions:
--
-- The following types have essentially the same structure, and this is
-- emphasised by the similarity of names.

   RPCStatusInfo          ::= [APPLICATION 110] IMPLICIT
                               SEQUENCE {RPCStatus,
                               RPCDiagnosticCode OPTIONAL,
                               RPCDiagnosticMessage OPTIONAL}

   RPCBindingStatusInfo   ::=  SEQUENCE {RPCBindingStatus,
                               RPCBindingDiagnosticCode,
                               RPCBindingDiagnosticMessage}
-- In both, the status distinguishes the first level of granularity within which the
-- diagnostics provide fine grained distinctions. The diagnostic code is intended
-- for interpretation by programs. The diagnostic message should be a natural
-- language equivalent of the diagnostic code, and is intended for human
-- readability.

   RPCEnvironmentalError ::= RPCStatusInfo
-- This type packages information about environmental errors. These errors are
-- distinguished from others by their status value (see below).
----------------------------------------------------------------------------------------------------
-- Second level definitions:

   RPCStatus ::= INTEGER {
-- Positive values are used for status resulting directly from execution of the
-- procedure.
--
-- All the other values (negative) report RPCEnvironmentalError conditions
-- arising from the call being to a remote procedure.

   normal                              (0),   -- this is the expected outcome.
-- The remote procedure has executed exactly once, produced RPCOutput (of
-- which this status value is part), and has returned reporting success.

   warning                             (1),
-- The remote procedure has executed exactly once, produced RPCOutput (of
-- which this status value is part), and has returned; but the output might not be
-- what the user expected. E.g. a floating point underflow has occurred and a
-- result has been set to zero.
```

```
   abnormal                                  (2),
-- The remote procedure has executed exactly once, produced RPCOutput (of
-- which this status value is part), and has returned; but an error has occurred
-- during the execution. The output produced is not all correct. To avoid
-- marshalling errors, incorrect output values may have been set to the NULL
-- type (or otherwise modified).

   error                                     (3),
-- The remote procedure has executed exactly once, but terminated without
-- RPCOutput. Instead an RPCError was produced (of which this status value is
-- part). The execution may have been incomplete (e.g. involuntary termination
-- due to an arithmetic overflow).

-- RPC Environmental Errors:

-- In the following text the term "executes at most once" means that the RPC
-- Service guarantees only that the remote procedure will not execute more than
-- once. Execution of it has or will occur once, or never, and may be complete
-- or incomplete.

   rOSEGeneralProblem                        (-1),
-- Corruption of Remote Operations protocol control information has been
-- detected at the Remote Operations Service. The details are defined in ISO
-- 9072/1. The remote procedure executes at most once.

   rOSEInvokeProblem                         (-2),
-- An unacceptable invocation has been detected at the Remote Operations
-- Service, locally or remotely (in the latter case it may be a protocol error). The
-- details are defined in ISO 9072/1. The  remote procedure executes at most
-- once.

   rOSEReturnResultProblem                   (-3),
-- An unacceptable RO-RESULT APDU has been detected as a Remote
-- Operations protocol error at the Remote Operations Service. The details are
-- defined in ISO 9072/1. The remote procedure executes at most once.

   rOSEReturnErrorProblem                    (-4),
-- An unacceptable RO-ERROR APDU has been detected as a Remote
-- Operations protocol error at the Remote Operations Service. The details are
-- defined in ISO 9072/1. The remote procedure executes at most once.
```

```
    interconnectionProblem              (-5),
```
-- The underlying interconnection is not available. This is detected at the
-- Association Control Service (ACSE). The details are defined in ISO 8649. The
-- remote procedure executes at most once.

```
    rPCBindingProblem                   (-6),
```
-- The RPC Binding is not available. This is detected at the D2 service. The
-- remote procedure executes at most once.

```
    crashProblem                        (-7)}
```
-- The called procedure (and/or the environment in which it executes) has
-- crashed. This has been detected and reported by some implementation
-- specific means, independently of the means inherent in the other status
-- values. The remote procedure executes at most once.

```
    RPCDiagnosticCode ::= INTEGER
```
-- An optional value which gives the definitive details of what happened to the
-- remote procedure call. The context within which it is interpreted is implicit in
-- the **RPCStatus** value, as follows:
--
-- If RPC status is **normal** the diagnostic code shall be zero (0).
-- If **RPCStatus** is one of **warning, abnormal** or **error**:

-- the interpretation of the **RPCDiagnosticCode** is either defined in the
-- specification of the D1 interface being used, or not (in which case it
-- may be application-specific or implementation-specific, according to
-- some other specification).

-- If **RPCStatus** is one of:

-- **rOSEGeneralProblem, rOSEInvokeProblem, rOSEReturnResults**
-- **Problem** or **rOSEReturnErrorProblem:**
-- the interpretation of the **RPCDiagnosticCode** is the ROSE problem
-- types defined in ISO 9072/1 and 9072/2.

-- **interconnectionProblem:** the interpretation of the
-- **RPCDiagnosticCode** is the rejection and abort types defined in
-- ISO 8649.

-- **rPCBindingProblem:** see **RPCBindingDiagnosticCode** below.

-- **crashProblem:** in this case the interpretation of the
-- **RPCDiagnosticCode** is implementation-specific and is not defined in
-- this standard.

```
RPCDiagnosticMessage ::= RPCCharacterString
-- An optional character string, describing in natural language the reason for the
-- returned status.
--
-- When the RPCStatus value is normal this string (if present) shall start with
-- "Normal Result" (or its equivalent where the language used is not English),
-- followed optionally by additional information. In such cases the diagnostic
-- information is logically redundant, but provides reassurance and readability
-- which may be useful when invoking independently supplied remote
-- procedures, particularly during testing.

RPCBindingStatus ::= INTEGER {
  bindingOK            (0),
  -- the RPCBindingDiagnosticCode shall be rPCBindingOK.

  rPCBindingProblem    (1)}
  -- the RPCBindingDiagnosticCode shall not be rPCBindingOK.

RPCBindingDiagnosticCode ::= INTEGER {
-- These are problems directly relating to the D2 interface.
  rPCBindingOK                                   (0),
  localNameProblem                               (1),
  rPCBindingClassProblem                         (2),
  serverNameProblem                              (3),
  serverLocationHintProblem                      (4),
  unknown-D1-Specification                       (5),
  d1-SpecificationNotSupportedLocally            (6),
  d1-SpecificationNotSupportedRemotely           (7),
  rPCVersionsIncompatibleAtD3-level              (8),
  securityProblem                                (9),
  resourceProblem                                (10)}

RPCBindingDiagnosticMessage ::= RPCCharacterString
-- A character string, describing in natural language the reason for the
-- returned status. It should correspond to the names defined in the
-- RPCBindingDiagnosticCode values (or their equivalent where the language
-- used is not English).

END -- end of ECMA Basic RPC Error Management definitions.
```

SECTION TWO : INTERCONNECTION STRUCTURE

6. APPLICATION SERVICE ELEMENTS

6.1 Introduction

A distributed application's remote procedure calls across the D1 interface are specified as an OSI Application Service Element (ASE).

The D1 notation is a subset of the Remote Operations notation defined in ISO 9072/1.

This notation for defining application-specific remote procedure calls is explained informally by the examples in Appendix F, which should be read at this point.

6.2 D1 Notation Definition

The D1 notation comprises:

a) the interface declaration structure defined in 6.3;

b) the ISO 9072/1 OPERATION macro and APPLICATION-SERVICE-ELEMENT macro, with the restrictions defined respectively in 6.4 and 6.5;

c) the ISO 8824 ASN.1 abstract syntax as visible via the interface declaration structure (a) and macros (b).

6.3 Interface Declarations

An interface declaration conforming with this standard:

a) shall be a complete ASN.1 syntax definition module conforming with ISO 8824;

b) shall identify the interface by an ASN.1 module name (an **RPCCharacterString** value);

c) should distinguish the interface declaration globally from all others (including all other versions of this remote application interface) by an ASN.1 Object Identifier which qualifies the module reference; and additionally by a comment "-- version xxxx --", immediatly after the DEFINITIONS keyword; where xxxx is an **RPCCharacterString** value;

d) shall define exactly one remote application interface;

e) shall use the APPLICATION-SERVICE-ELEMENT and OPERATION macros of ISO 9072/1 via an ISO 8824 IMPORT macro;

f) shall define the remote application interface via the APPLICATION-SERVICE-ELEMENT macro of ISO 9072/1;

g) shall make this Application Service Element available to other ASN.1 modules via an ISO 8824 EXPORT macro;

h) shall define one or more operations via the OPERATION macro of ISO 9072/1;

i) shall identify each OPERATION by a name and value that is unique within the declaration;

j) shall not directly use the ERROR macro defined in ISO 9072/1 (this is used indirectly, see clause 5);

k) shall use the LINKED Operations of the ISO 9072/1 OPERATION macro to specify any remote procedure callback;

l) shall not use the BIND and UNBIND macros defined in ISO 9072/1 (they are used at the underlying D3 interface);

m) should include comment text to explain the function and purpose of the interface, and the semantics and ordering constraints of its operations.

A subject for future study is use of methods more formal than (m) above for specifying operation semantics etc.

6.4 OPERATION Macro

The content of the OPERATION macro used in the D1 notation is restricted as follows:

a) the ARGUMENT of an OPERATION macro shall consist exclusively of elements within the **RPCInput** metatype structure defined in clause 4;

b) the RESULT of an OPERATION macro shall consist exclusively of elements within the **RPCOutput** metatype structure defined in clause 4;

c) the ERRORS of an OPERATION macro shall consist of one error name, **RPCError** defined in clause 4;

d) the integer value which distinguishes an Operation shall be non-negative.

Requirements (a), (b) and (c) are to facilitate language bindings. Requirement (d) is to allow future distinctions between application-specific operations and generic operations which may be needed to support future extensions to RPC, etc.

6.5 APPLICATION-SERVICE-ELEMENT Macro

The APPLICATION-SERVICE-ELEMENT name value should be the same as that of the ASN.1 module name.

The orientation of Operations shall be CONSUMER INVOKES.

The Operations themselves may have LINKED child Operations, allowing remote procedure callback.

7. APPLICATION CONTEXT

7.1 Introduction

The D3 RPC interface is specified as an OSI Application Context, in the ISO 9072/1 notation. This defines OSI interconnection details.

7.2 D3 Interface Specification

The following text through to the end of this subclause is the D3 interface specification, in ISO 8824 and ISO 9072/1 notation.

```
BasicRPC-D3-Interface DEFINITIONS -- version 1 -- ::=
BEGIN

-- This ASN.1 module specifies the D3 interface as an Application Context
-- conforming with ISO 9072/1.
--------------------------------------------------------------------------------------------
-- The names of the RPC Application Service Element and Application Context
-- referenced by this module are user-defined, application-specific,
-- implementation-dependent. Angled brackets <...> are used to indicate where
-- this module should be customised by substituting the appropriate names etc.
--------------------------------------------------------------------------------------------
IMPORTS    APPLICATION-CONTEXT, BIND, UNBIND -- macros
           FROM Remote-Operation-Notation -- ISO 9072/1.
           {joint-iso-ccitt remoteOperations(4) notation-extension(2)}


IMPORTS    <D1 ASE name>
           FROM <D1 module name>
           {objectIdentifierOf<D1 ASE name>}
        -- imports the particular D1 ASE from the nominated D1 specification
        -- (both names should be the same, see clause 6).


<Application Context Name> APPLICATION-CONTEXT
     APPLICATION SERVICE ELEMENTS      {aCSE}
     BIND                              ConnectRPC
     UNBIND                            DisconnectRPC
     REMOTE OPERATIONS                 {rOSE}
     INITIATOR CONSUMER OF             {<D1 ASE name>}
     ABSTRACT SYNTAXES                 {objectIdentifierOfAbstractSyntax1}
::= {objectIdentifierOf<this Application Context>}
-- The operations are all ISO 9072/1 Operation Class 1 (and may include
-- LINKED child operations), via Association Class 1.
--
-- Inclusion of other ASEs (e.g. CCR) is for future study.


ConnectRPC ::= BIND
DisconnectRPC ::= UNBIND
BIND ::= NULL
UNBIND ::= NULL
```

```
-- These ROSE BIND and UNBIND operations define the temporal scope of the
-- ACSE Association which is used for the Remote Operations (see ISO 9072/1).

-- As used here they do not exchange any parameter information between the
-- client and server. This restriction is consistent with procedure call language
-- semantics, and facilitates future use of interconnection and remote binding
-- techniques that do not necessarily include end-to-end handshakes before the
-- RPC interactions commence (with opportunity for parameter exchange).

END -- Basic RPC D3 Interface definitions.
```

8. SERVICE MAPPINGS

8.1 Introduction

The mappings from the D2 and D3 services onto underlying inter-connection services are illustrated in figure 5, and are defined in clauses 8.2 and 8.3 respectively.

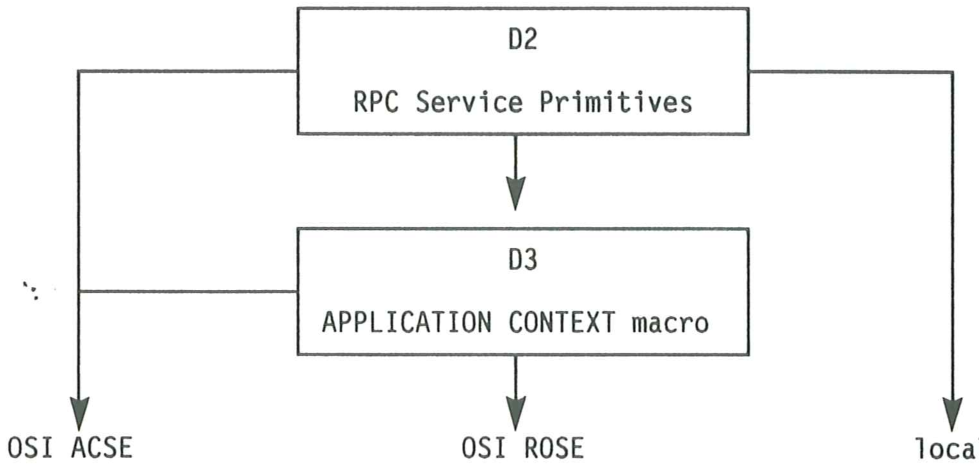


Figure 5 - Service mappings

The D1 interface used for a particular RPC interaction is inserted into the APPLICATION CONTEXT macro, as defined in 7.2.

8.2 Mapping from D2

8.2.1 Overview

The mapping from the D2 service to the underlying services is summarised in Table 1. It is provided by D2 mapping functions local to the client and the server.

D2 element name		mapping	subclause
BeginRPCBinding		---➤ BIND	8.2.2
GetRPCAttributes		---➤ local	8.2.6
RPCCall		---➤ OPERATION	8.2.5
EndRPCBinding		---➤ UNBIND	8.2.3
RPCProceduresOffered		---➤ local	8.2.6
RPCInvocation & RPCReturn		---➤ OPERATION	8.2.5
RPCStatusInfo or	)	◄--- A-ABORT	8.2.4
RPCBindingStatusInfo	)	◄--- A-P-ABORT	8.2.4
	(	◄--- RO-REJECT-U	8.2.6
RPCStatusInfo	(	◄--- RO-REJECT-P	8.2.6

Table 1 - Summary of D2 mapping

8.2.2 Binding

In this mapping the interconnection to support an RPC Binding is provided by an ACSE Association.

The parameters of D2 **BeginRPCBinding** identify the server in ways that may be dependent on the programming language, host environment and other implementation-specific details. The D2 mapping function converts these values to parameters of ACSE A-ASSOCIATE, in ways that are implementation-specific. Typically this would involve access to OSI Directory Services etc.

The D2 mapping function establishes the ACSE Association, using the D3 BIND operation. This is done at some time between the program being initialised and execution of the first D2 **RPCCall** or D2 **GetRPCAttributes** element. Success or failure of this Association establishment is reported via the return values of subsequently invoked D2 elements (see 8.2.4 and 8.2.7).

8.2.3 Unbinding

The D2 mapping function maps **EndRPCBinding** to the D3 UNBIND.

8.2.4 Disruptions

The ACSE association may be disrupted by A-ABORT and A-P-ABORT services, in the ways defined in ISO 9072/1. These events and any failure to establish the ACSE Association are reported via the **RPCStatusInfo** value of the current/next **RPCCall** element, or the **RPCBindingStatusInfo** value of the next **GetRPCAttributes** element, if any.

8.2.5 RPCCall

The D2 **RPCCall, RPCInvocation**, and **RPCReturn** service primitives model the structure of a remote procedure call. The D2 mapping is as follows.

a) Procedure Name. The **ProcedureName** component of the **RemoteProcedureReference** of the **RPCCall** primitive identifies the remote procedure, as specified below.

At the calling procedure the D2 mapping function maps this name to the corresponding D1 Operation name, defined in the D1 interface being used. The mapping is implementation specific. If the call is a remote procedure callback, the Operation name will be that of a linked child Operation of the Operation by which the calling procedure has itself been called.

At the called procedure the D2 mapping function maps the Operation name to the corresponding local **ProcedureName**. This mapping is defined by the **RPCProceduresOffered** primitive.

b) Values copied to the Called Procedure. At the calling procedure the D2 mapping function maps the **RPCInput** of the **RPCCall** primitive to the ARGUMENT of the D1 Operation.

At the called procedure the D2 mapping function maps this information in complementary fashion to the **RPCInput** of the **RPCInvocation.**

c) Values copied to the Calling Procedure. At the called procedure, depending on which outcome occurrs, the D2 mapping function maps **RPCOutput** to the RESULT of the D1 Operation, or **RPCError** to the ERRORS and error PARAMETER of the D1 operation.

At the calling procedure the D2 mapping function maps this information in complementary fashion to the **RPCOutput** or **RPCError** of the **RPCCall** primitive.

d) RPC Environmental Error. The **RPCEnvironmentalError** of the **RPCCall** primitive is provided by the D2 mapping function at the caller. The problem may arise from some local cause, or from the underlying D3 service (see 8.2.6 and clause 5).

8.2.6 Operation Rejections

The D2 mapping functions are the source and destination of the RO-REJECT-U and RO-REJECT-P service elements of ROSE. These are reported to the calling procedure via the **RPCEnvironmentalError** of the **RPCCall** primitive (the **status** values are "**rOSEProblem...**").

These rejections are not reported to the called procedure via the D2 service interface. This is inherent in the asymmetric structure of this Basic RPC, and is reinforced by the asymmetric protocol characteristics defined in clause 9.

8.2.7 Local Matters

The **RPCProceduresOffered** primitive is only of local significance (at the called procedure), and is mostly relevant at program construction time.

The **GetRPCAttributes** primitive is only of local significance in that it obtains reports from the local D2 mapping function.

8.3 Mapping from D3

The APPLICATION-CONTEXT macro of the D3 service is mapped onto the underlying OSI services as defined in ISO 9072/1.

9. PROTOCOL

9.1 Protocol Specification

The OSI services identified in clause 8 are mapped onto OSI protocols as defined in ISO 9072/2.

The use of the ISO 9072/2 Remote Operations protocol may be restricted as defined in clause 9.2.

9.2 Asymmetric Protocol

For the Basic RPC, it is desirable that the protocol for each call (or callback) is restricted to a simple request/response message pair, as illustrated in Figure 6.

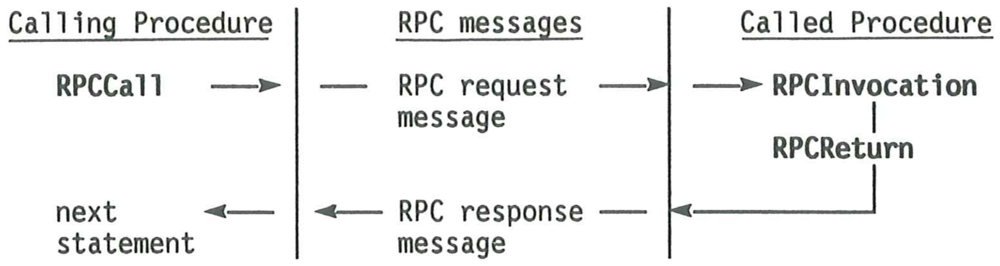


Figure 6 - RPC message pair interaction

In terms of the ISO 9072/2 Remote Operations protocol, the RPC request message is always an RO-INVOKE APDU, and the RPC response message is normally an RO-RESULT APDU, but exceptionally it is an RO-ERROR APDU or an RO-REJECT APDU; and it may be an RO-INVOKE APDU if there are LINKED child Operations (remote procedure callback).

ISO 9072/2 specifies that an RO-REJECT APDU be sent in response to an unacceptable APDU (ie. protocol error). This could result in an extended exchange of messages per remote procedure call (RPC request message, RPC response message, rejection of response, etc.). For the specialised asymmetric interactions of RPC, the server should not be required to handle this extended interaction (its RPC response message should be final, even if it is an ISO 9072/2 protocol error).

The restriction applied here to avoid this is that: if the client D3/ROSE function receives an unacceptable APDU, it should not send any RO-REJECT APDU and should abort the ACSE Association.

APPENDICES

APPENDIX A

PROCEDURE CALL TUTORIAL

This Appendix is not an integral part of the standard.

A.1 Introduction

As a preliminary step towards understanding procedure calls which are remote, this tutorial explains local procedure call characteristics. It assumes only a basic knowledge of programming techniques. This Appendix is intended to be complete in itself; therefore it repeats some information that is provided elsewhere in the standard.

The references to standards publications are in 1.3. Other references are in Appendix C.

A.2 General Structure

We are concerned here with the general language structure and execution structure of procedure calls. Language-specific details are avoided.

- Inter-module communication. The procedure call mechanism exists as a means of orderly communication between logically separate procedures of a sequential computer program. The participants in a procedure call are referred to here as the calling procedure and the called procedure.
- Procedure structure. A procedure is a closed sequence of instructions that is entered from and returns control to an external source. It consists essentially of a procedure definition and a procedure body which is terminated by some kind of return statement.
- Referencing. The called procedure is referenced by its procedure name. This value is necessarily unique according to the naming scope rules of the programming language concerned. The name of the calling procedure is not visible to the called procedure.
- Parameter passing. Information is communicated between the procedures by parameters which are passed by the call, as explained in A.4.
- Procedure definition. The identity and parameters of a procedure are defined in its procedure definition. In most languages this definition is at the beginning of the procedure itself.

- Sequential execution. Procedure call execution has a request/response structure with exact synchronisation which is fundamental to the semantics of procedure calls. It is explained more fully in A.3.
- Nesting. A called procedure may itself make procedure calls to other procedures, which may call others, etc. This calling may be nested to arbitrary depth. Languages have scoping rules which restrict what can be called.

Figure A.1 illustrates an example of a modular program with procedure call structure.

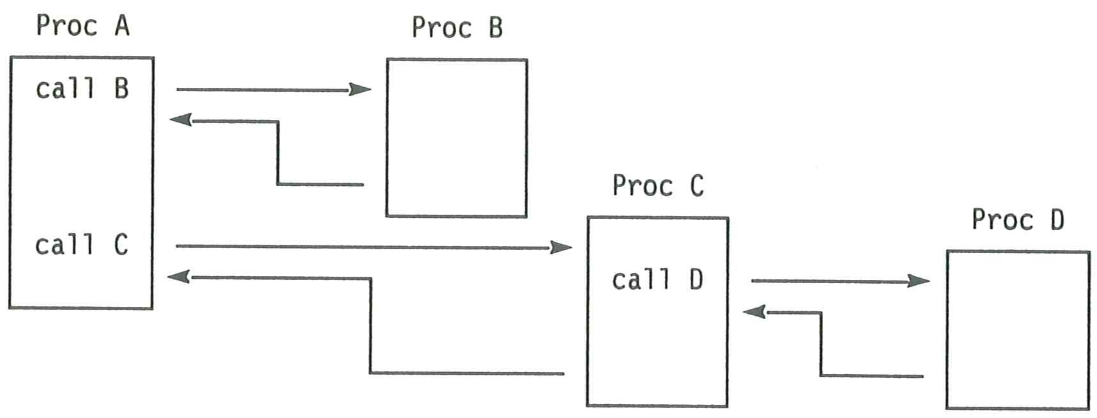


Figure A.1 - A modular program, structured into separate procedures integrated via procedure calls

A more complex example might include recursive calls in which a called procedure makes procedure calls to itself, to its caller, etc.

A.3 Computational Model

The computational model of procedure calls is now explained in more detail.

With respect to any particular procedure call there is one thread of execution control, as illustrated in Figure A.2 (a). The execution passes from the calling procedure to the called procedure, and then back again. The execution of the calling procedure resumes only after the called procedure has been executed.

At the called procedure the thread of execution may itself make further (nested) procedure calls in order to accomplish its purpose. This same procedure then has 'calling' and 'called' roles with respect to different procedure calls. E.g. the procedure x in Figure A.2 (b) is the called procedure with respect to the procedure call from the first procedure, and the calling procedure with respect to the procedure call to the third procedure.

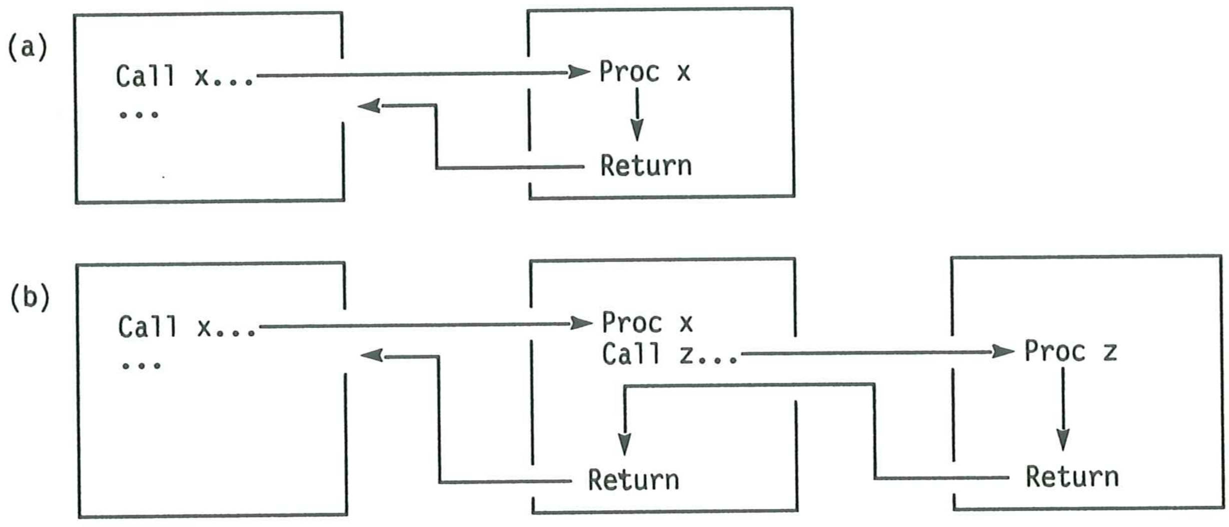


Figure A.2 - Procedure execution

The existence of procedure call linkage does not, of itself, extract specific obligations from the called procedure. The called procedure has general obligations inherent in being accessible to such linkage; but it does not know what linkage actually exists or is in active use until a procedure call uses it; nor is it obliged to remember this relationship outside the duration of the procedure call. The linkage only provides repeatable access to a predetermined destination (the called procedure).

The calling and called procedures should achieve any necessary continuity between successive procedure calls via the effects of the procedure calls themselves. This is done by the procedure calls referencing something that persists throughout this time (e.g. a reference 'handle' value, or a transaction identifier). This kind of binding is rather different from that familiar to OSI experts (ie. ACSE Associations). There are examples of it in F.3 and F.5.

A.4 Parameter Structure

Procedure calls exchange information explicitly via parameters. But in most languages there can also be hidden information flow between procedures via shared information which is accessible within a scope that is global to the procedures.

- Parameter passing semantics. The called procedure needs to be able to locate values of the parameters specified in the calling statement. This is usually done by passing the addresses of the parameters to the called procedure, and is termed "call by reference". Some languages, e.g. PASCAL, may pass values directly, and this is termed "call by value". A few languages use a further mechanism, termed "call by name". See [Gries]
- Parameter quantity. The procedure definition of a procedure fixes the number of parameters that can be passed to it.

- Formal parameters. The names, types and sequence of parameters are defined in the procedure definition. These are termed the 'formal parmeters' of the procedure.
- Actual parameters. The procedure call statement in the calling procedure identifies the locally accessible information corresponding to the parameters. These are termed the 'actual parameters' of the procedure call.
- Parameter direction. For each parameter there is a defined direction of information flow. A parameter which passes information into the called procedure is referred to here as an "input" parameter; vice versa an "output" parameter; or both directions "input/output".

These parameter passing concepts are summarised in Figure A.3.

```
   The Calling procedure              The Called procedure
 with the actual parameters        with the formal parameters
```

```
...
Call B (IN x,y,z): OUT p
...
...
```

```
Proc B
(IN m, n, q): OUT s
Begin
...
End
```

```
Mapping used for these parameter values in program execution:
                    x ──► parameter 1 ──► m
                    y ──► parameter 2 ──► n
                    z ──► parameter 3 ──► q
                    p ◄── parameter 4 ◄── s
```

Figure A.3 - Procedure Call parameter passing

In most languages the programmer is left to define output parameters for status information and error diagnostics. This is explained more fully in A.6. Some languages, such as Mesa [Mitchell], include comprehensive exception handling in which each procedure call nominates exception procedures to handle defined error conditions.

A.5 Implementation Considerations

Type checking. Compilers (and run time interpreters) may include type checking to ensure that the types of the actual parameters of the calling procedure are consistent with the types of the formal parameters in the procedure definition of the called procedure. Some languages guarantee that all procedure call parameters are "type complete" and "type safe"; e.g. Algol 68 [Wijngarden].

Linking. The object-code of the calling and called procedures may be constructed independently. Tying them together is termed linking. This is the process of binding the object-code to the addresses of the relevant procedures, parameters and variables. Depending on the language system and the environment, the linkage may be pre-configured (static or early binding), or is determined when or after the software is loaded for execution (dynamic or late binding).

Mixed languages. In some execution environments it is possible for procedure call linkage to exist between modules with source code in different languages, compiled to common target conventions.

IRDS. Master copies of procedure definitions may be stored in a data-dictionary (e.g. an Information Resources Dictionary System - IRDS). The definitions are then maintained there separately from the program procedures.

A.6 Error Management

When a procedure executes, various things can happen to affect the output produced by it. The classification below is used in the main body of the standard.

(a) Normal. The procedure's execution is successful, all the values are computed as expected.

(b) Warning. The procedure has produced output; but detected a situation that should be brought to the caller's attention. For example, the procedure had to change the value of an input parameter, ignored an input parameter in order to complete execution, set a result to zero after floating point underflow, encountered the end-of-file condition, etc.

(c) Abnormal. The procedure detects a severe error which prevents successful completion of the computation. Part of the output may have been computed; but the termination is orderly, in that the procedure is able to indicate that not all the output is present.

(d) Error. The procedure is unable to complete execution and return output values. E.g. it detects an error in the input parameters and terminates abruptly without computing any results; or it is abnormally terminated by an execution problem such as arithmetic overflow. In such cases it is often impossible to transfer output parameters correctly, because output values will not exist, and whatever occupies their storage may not have the correct representation for the output's type.

The first three situations should be handled by returning in the output parameters, along with any computed values, a status value. The fourth case (d) should be handled by returning a status value which indicates that parameters values are undefined. Typically this kind of situation (d) is detected and reported by the language execution system rather than the called procedure itself.

In some implementations, the occurrence of problems such as (c) and (d) may lead to the calling procedure being aborted without it being aware of them.

A.7 Summary

Local procedure call characteristics of particular relevance are:

- Asymmetric. A procedure call interaction has a strict request/response discipline, and occurs at the initiative of the calling procedure.
- Synchronous. Execution is synchronous. Execution of the calling procedure waits until the return from the called procedure.
- Restricted Parameters. The information explicitly communicated via procedure calls is in parameters of defined quantity, sequence and type.
- Variability. Some details of procedure call structure are different in different languages and execution environments; but there is a core that is common to nearly all.
- Software engineering. Languages and software development processes may include highly developed structure to support procedure calls and related software modularity.

These procedure call characteristics of programming languages are the principal determinants of RPC structure.

APPENDIX B

RPC TUTORIAL

This Appendix is not an integral part of the standard.

B.1 Introduction

This tutorial explains Remote Procedure Call (RPC) concepts. It assumes a basic understanding of procedure call structure as explained in Appendix A, and of current OSI standardisation. It assumes no prior knowledge of RPC. This Appendix is intended to be complete in itself; therefore it repeats some information that is provided elsewhere in the standard.

The references to standards publications are in 1.3. Other references are in Appendix C.

As explained in [Birrell 84], the idea of remote procedure calls is simple. It is based on the observation that procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across communication networks.

When a remote procedure is invoked, execution of the calling procedure is suspended, the parameters are passed across the network to the remote environment, and the called procedure is executed there. When the called procedure terminates and produces its output, this is passed back to the calling environment, where execution of the calling procedure resumes as if returning from a single-machine call. While the calling procedure is suspended, other processes on that machine may still execute (depending on the details of the parallelism of that environment and the RPC implementation).

There are many attractive aspects to this idea. One is clean and simple semantics: these should make it easier to build distributed computations, and to get them right. Another is efficiency: procedure calls seem simple enough for the communication to be quite rapid. A third is generality: procedure call is already the most important mechanism for communication between the logically separate parts of software systems. A more general point is that an RPC approach helps to assure that applications investment by users is network- independent.

RPC concepts first became generally visible in 1976 [White], and were integrated into a proprietary networking architecture in 1981 [Xerox].

Since the early 1980s RPC techniques have been thoroughly evaluated and reported by the research community; e.g. in [Birrell 84], [Birrell 85], [Gibbons], [Hamilton], [Jones], [Nelson], [Panzieri]. Their general conclusion is that RPC is a vital ingredient of distributed interactive processing.

Since the mid 1980s proprietary RPC facilities have become available on various operating systems, e.g. on Unix [Sun] and on PC-DOS [IBM].

In 1984 CCITT Rec. X.410 established a Remote Operations notation and protocol, now widely used for open standards. This was evolved from an RPC design [Xerox], but does not claim to be an RPC. However, comprehensive RPC characteristics are latent in it (and are exploited in this standard).

B.2 Application Systems

B.2.1 RPC Application Characteristics

RPC is oriented to distributed applications in which there is interactive communication, program to program, with short response times and relatively small amounts of data transfer.

A remote component of an application is usually modelled as a named and complete service. Where access is via RPC, the primitives of the service are implemented as a corresponding family of remote procedures. This family of procedures representing the service is a remote program, as illustrated in Figure B.1.

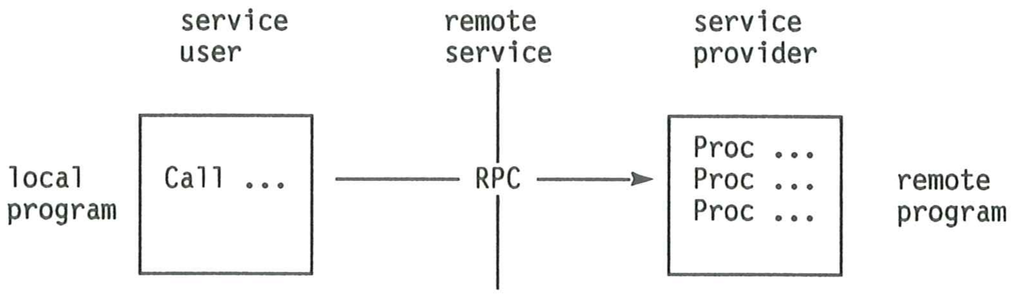


Figure B.1 - General structure

The remote service/program accessed via RPC may be application-specific (e.g. part of some financial application), or a generic shared resource (eg. a file server). The application system structure may be modelled according to the ECMA TR/42 framework.

B.2.2 RPC Remote Interface Declarations

The remote interactions are defined in an *interface declaration*. This is essentially a set of (abstract) procedure definitions of the remote service/program, combined with other relevant specification information. It defines the remote interface between the user and provider of the remote service. See Figure B.2.

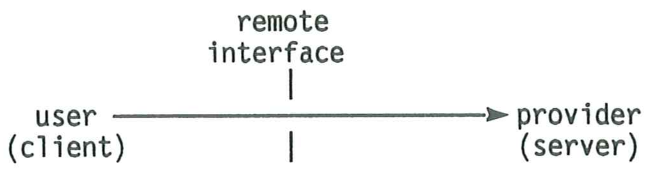


Figure B.2 - A Remote Interface

At this level of abstraction the parties to request/response interactions are usually termed *client* and *server*, as in ECMA TR/42. Use of client/server/ service terminology is not particular to RPC.

As explained in B.3.4, the interface declaration should be expressed in an *interface specification language* which allows heterogeneous implementations of the programs which provide and use the declared interface.

An interface specification language defines and enforce generic rules which would restrict call structure and parameter structure in ways generally applicable to heterogeneous RPC.

B.2.3 RPC Bindings

A server declares information about the services (procedures) it offers, and a client (directly or indirectly) makes use of such information about the services which it intends to use. This information is in terms of interface declarations, service names, qualitative controls, etc. Typically such information is stored in and retrieved from data-dictionaries (IRDS) and directories in ways general to all distributed processing (and not specific to RPC).

The term *RPC Binding* is used here for the access linkage which enables the client to access the server. It has essentially the same properties as local procedure linkage. Therefore, as explained in A.3, any binding to remote state information specific to the interaction is necessarily via parameters of the procedure calls themselves (and is not part of the RPC Binding mechanism). This kind of binding is inherent in the examples in F.3 and F.5.

B.3 RPC Considerations

B.3.1 Remoteness

We are concerned here with direct implications of the procedures being in physically separate computers linked by telecommunications.

- Naming scope. The naming scope of procedure calls is usually restricted to procedures in the same machine. Some extra naming provisions are needed to call external procedures.
- Binding and loading. In most languages the binding between procedures is implicit in the program structure. It occurs as a normal consequence of program composition and compilation. Remoteness implies some need for ex-

plicit binding action under program control at run time. Specific provisions may also be needed to load the remote procedure when it is needed. This is analagous to the way programs bind to files at run time and cause external magnetic media volumes to be mounted and dismounted.

- No shared memory. Local procedure calls depend, implicitly or explicitly, on the use of shared variables which can be accessed by the calling and called procedures. By definition, RPCs have no shared memory: the two procedures are usually in physically separate computers. The general conclusion for RPC is that formal parameters can not include pointers to off-stack memory locations, and that the parameter values must be copied from machine to machine (even where the normal compiler generated object code for a corresponding local call would be call by reference). The ideal parameter passing semantics for RPC are therfore call by value.
- Communications. The underlying support system necessarily uses communications as part of the mechanisation of the remote procedure calls. But this is not directly visible in the procedure definition, nor to the procedures using RPC.
- Error Management. Remoteness also introduces different error possibilities, as explained in B.3.3.
- Degree of remoteness. Much of the early experience with remote procedure calls was with local area networks. RPC implementation experience with wide area networks confirms that RPC techniques scale up to arbitrary degrees of remoteness. The ability to scale down to low degrees of remoteness (e.g. RPC across a backplane bus) is inherent in the derivation of RPC from local procedure call techniques.
- Security. Remoteness also has many security implications.

A preliminary conclusion to be drawn from the above is that a remote procedure call cannot be exactly like a local procedure call. This is confirmed by the other factors now considered.

B.3.2 Distribution Transparency

A major system and application design issue is whether or not to hide distributedness and its consequences. The term *distribution transparency* is used here for discussing the visibility of distributedness within distributed systems.

- Arguments for transparency. It can be advantageous if all the consequences of distribution are made transparent (ie. invisible). This hides complexity, simplifies the task of application designers, and enhances the re-usability of application code. The evolution of existing products based on centralised systems is then inherently straightforward. A successful experiment with such transparency for procedure calls is Unix United [Brownbridge].
- Arguments against transparency. Full transparency, which completely conceals distribution, can be relatively expensive in terms of underlying implementation effort and performance overheads. Moreover, it denies designers

the opportunity to exploit the consequences of distribution via decentralisation and replication of control, or data, or both.

System design choices lead to different transparency requirements, and full distribution transparency is not always necessary. Therefore standards should not pre-empt these choices.

In this Basic RPC standard the visibility of distributedness at the user programmer interface is confined to particular parameters of the RPC service primitives. The user program can choose to ignore these.

B.3.3 Reliability

We are concerned here with the reliability implications of the calling procedure and called procedure being in physically separate machines.

- Independent failures. With a local procedure call there is one process (ie. one execution context, one thread of execution) which either crashes or survives. But for a remote procedure call the process in one machine may crash while that at the other remains intact. This can lead to situations such as: remotely called procedures that are 'orphaned' by calling procedure crashes; calling procedures which become 'bereaved parents' waiting for replies from remotely called procedures that have crashed; and recovery situations in which the caller is uncertain whether the remote procedure has been executed (e.g. when network partitioning occurs during a remote interaction).
- Communications failures. The underlying communications system should hide the occurrence of communications failures, but cannot hide any prolonged inability to communicate (network partitioning).
- Compatibility. The physical separation of procedures emphasises the compatability and version control problems latent in all modularity.
- Error management. Error possibilities particular to remote and heterogeneous interactions require special error management. See B.4.2.

It should be appreciated that the most difficult reliability issues arise not from communications failures, but from host overloads, host crashes and remote application failures. Perfect communications would not remove these problems, therefore the solution is not just a matter of using reliable communications connections etc. See [Saltzer].

An RPC system should have appropriate execution reliability semantics. This subject is explained in [Panzieri]. The following choices of guarantees may occur: remote execution occurs *exactly once*; remote execution occurs *at most once* (including possibilities of no execution and incomplete execution); remote execution occurs *at least once* (including possibilities of multiple complete and incomplete executions).

The exactly once semantics are the most difficult to guarantee, and the at least once semantics are the easiest. The degree of difficulty affects the amount of RPC protocol etc. needed to provide the guarantees.

The remote execution reliability guarantees inherent in the OSI Remote Operations standards (ISO 9072/1 and 9072/2) are exactly once for interactions returning to the calling procedure normally without error, and at most once for all other outcomes. This seems to be the most appropriate choice for general OSI use (although other choices should not be permanently precluded).

B.3.4 Heterogeneity

Much of the experience with using RPC has been in homogeneous environments; ie. both ends use the same language and operating system.

An RPC system for OSI standards purposes must allow for heterogeneous environments: there will typically be different languages and different operating systems at each end.

These heterogeneity problems are now well understood after several years of practical experience with RPC in heterogeneous environments, e.g. Matchmaker [Jones] and HRPC [Bershad]. Powerful software tools systems have been developed to support RPC in heterogeneous environments, e.g. the stub generator in [Gibbons].

Some of the problems inherent in this heterogeneity are:

- Semantics. In different languages there are differences of semantics for what might seem to be the same data structures.
- Language characteristics. The data structures and procedure call structures supported in any one programming language are not all supported in exactly the same way in all other languages.
- Concrete syntax. Different machines use different conventions for the bit representation of data values, e.g. different byte orderings, size limits and floating point formats.

The basis for solving these problems is to specify the remote interactions in a canonical form which is independent of such variability. This requires use of an *interface specification language* and an *external data representation* which are independent of the choices of programming languages, operating systems, computers and networks, as fully explained in [Gibbons]. The next step is to define a *language binding* for each programming language of interest. This defines the mapping of the semantics and syntax of the programming language onto the RPC control structure and the external data representation structure.

B.4 RPC Structure

B.4.1 Call Structure

For the reasons discussed in B.3, remote calls via a heterogeneous RPC system need to be in some respects different from local procedure calls. However, they also need to be well integrated into the local language, software tools system and execution environment. The main points are:

- Language structure. The control structure and syntax of remote calls must be acceptable in the local language environment. Typically remote procedure calls will be implemented via a normal procedure call in the host language.
- Call destination. The immediate destination of the call is likely to be a local RPC support procedure (typically an 'RPC stub procedure') which handles the remote interaction. The ultimate destination is a particular procedure in a particular server which is remote from the calling procedure.
- Control structure. As with local procedure calls, execution of the calling procedure waits until the called procedure replies, or until this is pre-empted by an exception condition (see B.4.2).
- Actual Parameters. The call statement will define the actual parameters (in ways specific to the particular language binding).

These parameters should be consistent with the formal parameters of the interface declaration.

B.4.2 Marshalling

At the calling procedure, parameters that are input to the remote call are converted into the agreed external data representation, and are assembled into a protocol octet string for transmission.

At the called procedure, the protocol octet string is disassembled, and the value of each parameter is converted into the local data representation and put in the appropriate location in the called procedure's address space. Execution of the called procedure is then dispatched.

After the called procedure completes execution, the same process occurs in the opposite direction.

This process of converting, copying, assembling and disassembling RPC parameter values is termed "marshalling". It is illustrated in Figure B.3.

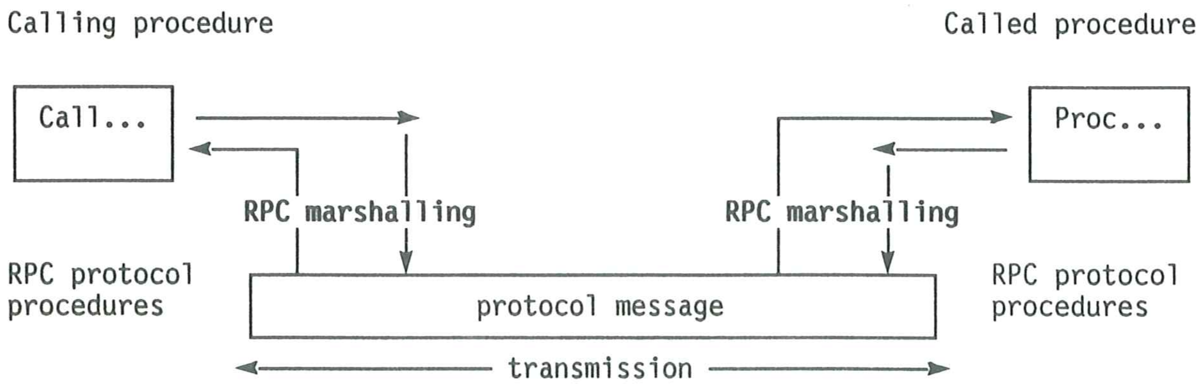


Figure B.3 - RPC Marshalling

In distributed systems the parsing, copying and converting of data for protocol interactions is often a major source of implementation complexity and of performance overheads. The systematic structure of RPC marshalling is amenable to automatic code generation (see B.5) and to specialised performance optimisations.

Where the client and server systems are homogeneous and have the same internal data representation, there is an opportunity for the parameter values to be transferred in native mode, with reduced marshalling overheads.

B.4.3 RPC Error Management

An RPC system should have comprehensive error reporting provisions. Most RPC systems distinguishes between:

- execution errors which arise from the procedure call itself (as explained in A.6); and
- environmental errors which arise from the called procedure being remote. Examples of environmental errors are: remote system crash, unrecoverable communications problems, naming and binding problems, security problems, protocol or syntax compatibility problems, etc.

A general way of distinguishing between these different error cases is to include in the procedure call definition extra parameters for status and diagnostics. This information would usually be examined by the calling procedure after the return, and could be used to decide what (if any) recovery action is necessary.

Another error management issue is the use of timeouts. The well researched solution adopted in [Birrell 84] is for the RPC system not to have user-visible timeouts. The client application program should have the usual kind of global timeouts for detecting all non-terminating execution (local or remote). This is consistent with the client responsibility for recovery. Internal to the RPC system there may be hidden timeouts for protocol error management, and for periodic checks that unusually long running calls are still executing.

B.4.4 RPC Process Structure

RPC does not require specialised process structures: implementations use whatever are the normal kinds of process structures for their language systems and execution environments. The general case is a single thread of execution per program. That is largely why this standard restricts the concurrency level within an RPC Binding to be one.

Where a program requires concurrent remote interactions this may be achieved via multiple programmed asynchronous interactions across message read/write interfaces to the underlying protocol machines. But the remote interactions then depart from procedure call language semantics.

Procedure call structure is fully preserved if the language/systems structure allows program execution to spawn multiple processes capable of executing asynchronously and subsequently resynchronising together. These constructs are usually termed a 'process fork' and a 'process join'. The separate asynchronous and logically concurrent threads of a computation can then make multiple concurrent procedure calls without directly delaying progress of the whole computation.

This kind of concurrency, built into the language/execution environment, is different from that traditionally used for interworking between data processing systems. Programming is simplified by using well-formed language constructs (fork and join), and protocols are simplified by decomposition into multiple logically separate interactions, each with a simple synchronous structure. There are also potential performance gains through opportunities for specialised local hardware and software support for handling the asynchrony and synchronisation internally, instead of across the network.

B.4.5 RPC Protocols

Protocols to support RPC consist mainly of request/response message pairs. The request message transfers the procedure invocation and the input values, and the response message transfers the output values (or is an error management message). Simple disciplines for sequence control, flow control and error management are inherent in this strict request/response structure. Each RPC protocol message (PDU) may have a complete self-identifying structure.

Connectionless communications services may be used to carry such protocols. But the usual practice is to introduce RPC techniques by using existing connection-oriented communications, and to evolve later to selective use of connectionless services where appropriate. The [Xerox] and [Sun] RPCs have followed this evolutionary course. Another possibile evolution is to carry the RPC data structures over specialised protocols optimised for process-to-process communication, as in [Birrell 84] and the Rex protocol [Otway] evolved from it.

B.5 RPC Software Engineering Process

An RPC system may be integrated into comprehensive software engineering processes which exploit the structuredness and relative simplicity of RPC.

The first stage of the process is *interface definition*. The main ingredient here is the interface specification language, together with means of documenting and maintaining interface declarations, ideally in machine processable form. This could include use of an Information Resources Dictionary Systems (IRDS), and software tools for syntax checking, specification animation, etc. This is an iterative process, as illustrated in Figure B.4.

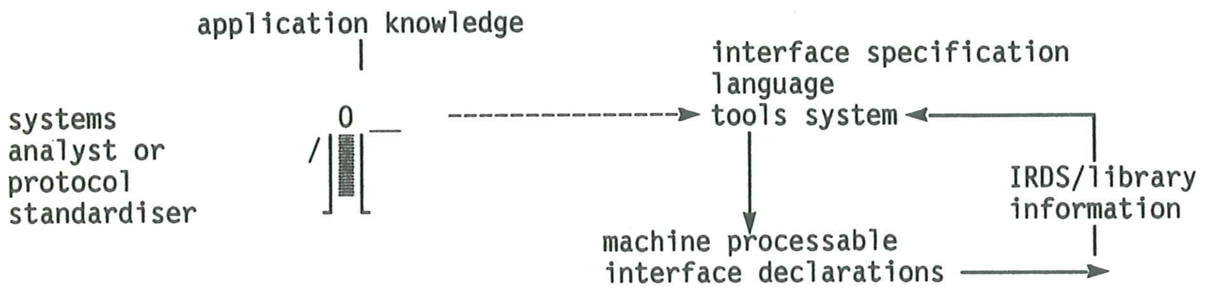

application knowledge
systems analyst or protocol standardiser
interface specification language tools system
IRDS/library information
machine processable interface declarations

Figure B.4 - Interface definition system

The next stage of the process is stub generation. This takes an interface declaration and generates the RPC procedures to marshal and unmarshal the parameters of procedure calls using that interface specification. It may also produce the appropriate generic procedures for handling RPC binding primitives for that interface, and code for the RPC buffering and the RPC protocol machine. The stub at the server end also includes a dispatching procedure which distinguishes between calls to different procedures within the server program.

Figure B.5 illustrates a highly automated system for stub generation which is fully described in [Gibbons].

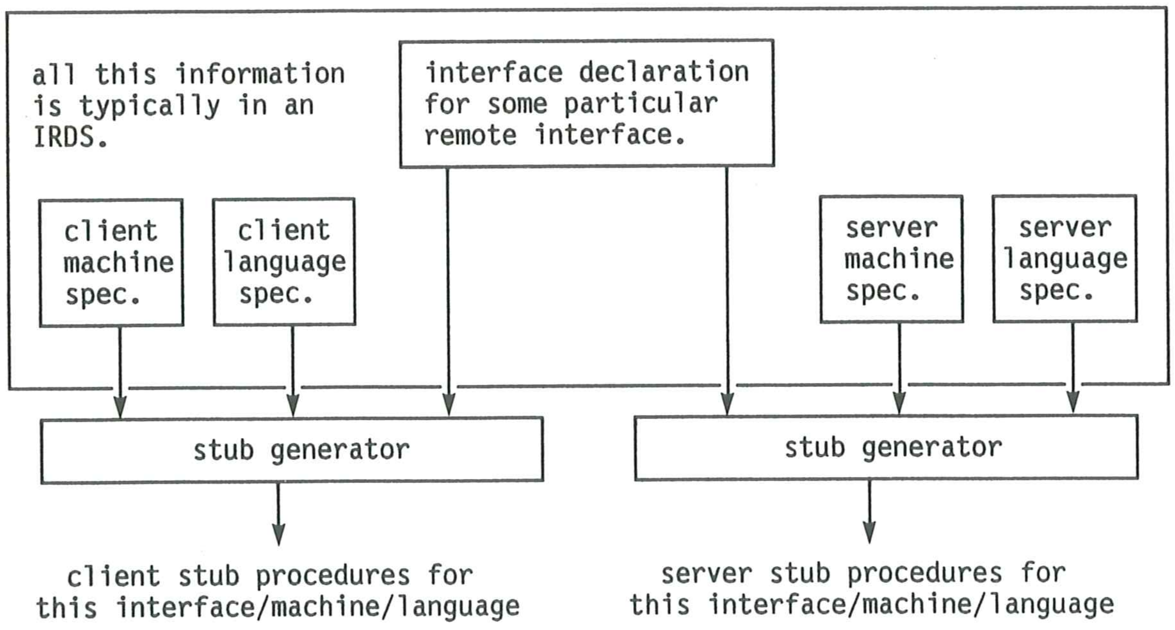

all this information is typically in an IRDS.
interface declaration for some particular remote interface.
client machine spec.
client language spec.
server machine spec.
server language spec.
stub generator
stub generator
client stub procedures for this interface/machine/language
server stub procedures for this interface/machine/language

Figure B.5 - An automated system for stub generation

The final stage of the automated development route brings together the application procedures using the RPC system and the generic stub code for the remote interfaces concerned. Program development at the client and server ends typically occurs independently (and possibly many times over, and in many separate business enterprises which use and provide the defined service). Automated development routes, based on the same machine processable text of the interface declaration, can assure a high probability of correct and compatible implementations. This may also reduce the amount and cost of validation and conformance testing.

This kind of RPC automation is also relevant for the construction of distributed application's software where the user-visible languages do not have procedure call constructs (e.g. Fourth Generation language systems).

B.6 RPC Summary

The essence of RPC is summarised as:

- program-to-program interworking;
- via programming language constructs;
- maximal de-coupling from communications matters;
- minimal intrusion into the user program source code.

There is consequent scope for automated development routes and specialised design and implementation.

APPENDIX C

BIBLIOGRAPHY

This Appendix is not an integral part of the standard.

[Bershad]	BN Bershad, DT Ching, ED Lazowska, J Sanislo, M Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems, IE Transactions on Software Engineering, Vol. SE-13, No. 8, August 1987.
[Birrell 84]	AD. Birrell and BJ. Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, vol 2, no. 1, pp 39-59, Feb 1984.
[Birrell 85]	AD. Birrell. Secure Communications using Remote Procedure Calls. ACM Transactions on Computer Systems, vol.3, no. 1, pp1-14, Feb.1985.
[Brownbridge]	DR. Brownbridge, LF. Marshall and B. Randell. The Newcastle Connection or UNIXes of the World Unite! Software Practice and Experience, 12 (12), 1147-1162 (December 1982).
[Cristian 82]	F. Cristian. Robust Data Types. Acta Informatica 17, pp 365-397 (1982).
[Gibbons]	PH. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. IE Transactions on Software Engineering, Vol. SE-13, No. 1, Jan 1987.
[Gries]	D. Gries. Compiler Construction for Digital Computers. John Wiley 1971. ISBN 0-471-32776-X.
[Hamilton]	KG. Hamilton. A Remote Procedure Call System. Ph.D dissertation, Computer Laboratory, University of Cambridge, UK.
[IBM]	Server Requestor Programming Interface (SRPI).
[Jones]	MB. Jones, RF. Raschid, MR. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. Procedings 12th. ACM Symposium on Principles of Programming Languages, Jan. 1985.
[Mitchell]	JG. Mitchell, W. Maybury, R. Sweet. Mesa Language Manual (Version 5.0). Tech. Rep. CSL-79-3, Xerox Palo Alto Research Centwe, Palo Alto, Calif.1979.

[Nelson]	BJ. Nelson. Remote Procedure Call. Ph.D dissertation. Department of Computer Science, Carnegie Mellon University, Pittsburg, Pennsylvania. Tech. Rep. CMU-CS-81-119, 1981.
[Otway]	D. Otway. Rex: a model for process to process interactions. Online Open Systems Conference. March 1987.
[Panzieri]	F. Panzieri, SK. Shrivastava. Rajdoot: a remote procedure call mechanism supporting orphan detection and orphan killing. Technical Report 200. Computing Laboratory, University of Newcastle upon Type.
[Saltzer]	JH. Saltzer, DP. Reid and DD. Clark: The End to End Argument. ACM Transactions on Computer Systems, Vol.2 No. 4, November 1987, pp 277-288.
[Sun]	Network File System: Remote Procedure Call Protocol Specification. Sun Microsystems Inc. 1984.
[White]	JE White. A High Level Framework for Network Based Resource Sharing. AFIPS Conference Procedings, NCC, 45: pp 561-570 1976.
[Wijngarden]	A. Van Wijngarden (ed). Report on the Algorithmic Language Algol 68. Mathematisch Centrum, Amsterdam. MR 101.
[Xerox]	Courier: Remote Procedure Call Protocol. Xerox Corporation, Stamford, CT, USA. Xerox Systems Integration Standard 038112, Dec. 1981.

APPENDIX D

FUTURE EXTENSIONS

This Appendix is not an integral part of the standard.

D.1 Introduction

This clause describes some potential extensions to the basic RPC protocol, and their language semantics and mappings onto ROSE and other OSI Application Service Elements.

In this standard the dialogue structure via the RPC binding between a client and a server has been restricted in the following ways:

- asymmetric - procedure calls are initiated in the direction client to server;
- request/response - each remote procedure call always consists of an end-to-end handshake (an invocation and its terminating response);
- no concurrency - each RPCCall must be completed before the next is invoked (although concurrency across multiple RPC bindings is not precluded);
- synchronous - where there is no concurrency, actions are one at a time, therefore synchronous (except that environmental failures are asynchronous).

These restrictions also apply when there is nested callback within a procedure call. To whatever depth the callback is nested, it is still serial request/response activity, and the outermost call is always invoked by the client.

These restrictions are made here because they are inherent in the procedure call semantics of most programming languages. Furthermore, these dialogue structures are sufficient for many kinds of distributed applications (although certainly not all).

In terms of the ISO 9072/1 and 9072/2 Remote Operations standards, the above restrictions are use of Operation Class 1 (including linked child Operations) within Association Class 1. The Remote Operations standards handle more complicated dialogue structures via Operations Class 2, 3, 4 and 5 and Associations Class 2 and 3. Thereby some "extensions" of the protocol subset used in this standard already exist.

The more complex dialogue structures generally require a more sophisticated model of computation than the procedure call. This is the subject of ongoing ECMA work on Open Distributed Processing (ODP) standardisation.

D.2 Concurrent Calls

By introducing concurrency into the computational model, there could be more direct support for concurrency of procedure calls between a client and a server.

These concurrent calls, asynchronous with respect to each other, might be mapped onto Remote Operation Class 2 within a single Association. This is for future study.

D.3 Calls with Immediate Return

In some distributed applications there are one-way interactions with no output parameters.

In such cases, the calling procedure would continue execution immediately after making the call, but without confirmation that execution of the remote procedure occurred and was successful.

This could be mapped onto Class 5 Operations. Similarly for the intermediate cases where only success or only failure is reported; they could be mapped onto Class 3 and Class 4 Operations.

But this is a departure from the language semantics of procedure calls, where return means that execution of the called procedure took place. In terms of distributed systems structure, these one-way interactions also tend to have producer/consumer structure, which is different from the client/server structure natural to RPC.

D.4 Mutual Calls

Two components of a distributed application may want to interact with each other via independent calls which are mutual calls, not nested callback.

This standard does not preclude mutual calls, but they would use separate RPC Bindings (mapped onto separate ACSE Associations, initiated in the opposite directions).

If such mutual calls are to be mapped onto a single association, the D1 interface might be extended to include bi-directional interfaces. Or this might be a hidden optimisation at the D3 interface level. Such interactions could be mapped onto the Remote Operations Association Class 3.

D.5 Interruption Call

It may be desirable for the caller to interrupt the execution of a remote procedure. For example, if the remote execution seems to be non-terminating, or if the caller no longer wants what was originally requested.

This extension requires the notion of signalling and exception handling to be included in the model of computation.

A drastic way of signalling interruption of a call would be by aborting the ACSE Association. This would cause an A-U-ABORT indication to be received by the component containing the called procedure. The local environment could then interrupt the procedure execution.

Another possible solution would be to signal the interruption asynchronously within the Association via a Class 2 Operation.

D.6 Orphan Detection

The called procedure is termed an "orphan" if it becomes unintentionally isolated from the calling procedure. This can occur where the calling procedure (the "parent") fails, or if there is prolonged communications failure (network partitioning).

The orphan has nowhere to send its output parameters, and may be uncertain what to do with any partial results which it retains.

The parent procedure may not remember what has happend when it recovers. Continued undetected existence of the orphan(s) may cause difficulties when the parent makes further calls. See [Panzieri].

When a procedure is orphaned, the component containing the orphaned procedure may detect this as a break in the ACSE Association (visible at the D3 interface as an A-P-Abort). But this detection mechanism requires communication to be connection-oriented, which may not always be the case in future. Another possible approach would be use of the OSI CCR ASE.

The computational model needs to be extended to describe the action taken by the orphan. In the simplest case, the orphan completes execution of the remote procedure but the return parameters are not returned to the calling procedure.

D.7 Multi-endpoint Interactions

Distributed applications may include multi-endpoint interactions in which an event at one component triggers action at several other components.

This is outside the scope of the current Remote Operations standards, and a multi-endpoint procedure call does not have normal procedure call semantics.

But linguistic structure for this kind of requirement is relevant to a general computational model for Open Distributed Processing (ODP).

D.8 Optimised Protocols

There may be requirements for standardisation of protocols optimised for efficient support of the highly specialised kind of request/ response interactions typical of RPC.

The current generation of OSI protocols are necessarily multi-purpose.

The expectation is that the ISO 9072/2 Remote Operations Protocol used for this ECMA Basic RPC could be mapped onto alternative protocol infrastructure, without change to this standard. Therefore the stability of this Basic RPC protocol is not threatened by such changes.

D.9 Total Operations

There may be requirements to exploit the explicit ERRORS structure of the Remote Operations notation more fully than in this standard. The aim would be to achieve the Total Operations structure and Robust Type characteristics explained in [Christian 82].

This is a problem because of inherent differences between this style of exception handling and the procedure call semantics and structure of most programming language. Therefore, it is left for future study.

D.10 Parameter Types

There may be requirements for RPC to support parameter types other than those included in this standard (e.g. more complex data structures, such as ODA document items).

The current standard includes most of the parameter types supported in ISO standard programming languages. There may be language binding problems for other types, and their direct use may be outside the scope of this standard. Extensions to the parameter typing are for future study.

APPENDIX E

REGISTER OF CODES

This Appendix is an integral part of the standard.

E.1 Operation Numbers

< 0	Reserved.
$\geqslant 0$	Available for user defined OPERATION macros in D1 interfaces, and not subject to any further allocation controls here.

E.2 Error numbers

< -1	Reserved.
-1	Used for **RPCError** (see clause 4).
$\geqslant 0$	Reserved for possible future use in user defined ERROR macros in D1 interfaces.

E.3 Tags

The encoding of RPC parameter lists which is defined in this standard uses the ASN.1 Tag type in particular ways. The following tag values are allocated to specific uses in this standard:

[APPLICATION 100] Tag for **RPCComplex**.

[APPLICATION 101] Tag for **RPCNumericString**.

[APPLICATION 102] Tag for **RPCProcedureName**.

[APPLICATION 103] Tag for **RPCArrayDescriptor**.

[APPLICATION 104] Tag for **RPCArrayBounds**.

[APPLICATION 105] Tag for **RPCBoundForDimension**.

[APPLICATION 106] Tag for **RPCArray**.

[APPLICATION 107] Tag for **RPCRecordDescriptor**.

[APPLICATION 108] Tag for **RPCVaryingStringDescriptor**.

[APPLICATION 109] Tag for **RPCStringWithBoundsDescriptor**.

[APPLICATION 110] Tag for **RPCStatusInfo**.

Use of application tag values <100 and >110 is for future study.

APPENDIX F

GUIDELINES FOR APPLICATION PROTOCOL DESIGNERS

This Appendix is not an integral part of the standard.

F.1 Introduction

This Appendix provides guidelines (through examples) for application protocol designers specifying D1 interfaces. These examples are not meant to suggest services which are necessarily desirable; their purpose is to explain the D1 notation.

Each example is a complete D1 interface declaration, including tutorial comment.

F.2 Date-Time Server

```
DateTimeService DEFINITIONS -- version 1 -- ::=
   BEGIN
------------------------------------------------------------------------------------------------

-- (The module begins with a statement of what it does and a declaration of its
-- relationships to other modules.)
------------------------------------------------------------------------------------------------

-- This ASN.1 module is the D1 specification for a simple time of day service.
-- The service supports a single operation (procedure), GetDateTime, which
-- returns the current date and time in UTC time format.


-- To make the Application Service Element available to other ASN.1 modules:
   EXPORTS dateTimeService;


-- External definitions used:


   -- all D1 modules have this import:
   IMPORTS OPERATION
           FROM Remote-Operation-Notation      -- ISO 9072/1
                {joint-iso-ccitt remoteOperations(4) notation(0)};
```

```
-- all D1 modules have this import:
IMPORTS APPLICATION-SERVICE-ELEMENT
 FROM Remote-Operation-Notation-extension      -- ISO 9072/1
             {joint-iso-ccitt remoteOperations(4)
              notation-extension(2)};

-- all D1 modules have this import:
IMPORTS RPCStatusInfo
        FROM ECMABasicRPC-ErrorManagement;
        -- clause 5.2 of this Standard

-- all D1 modules have this import:
IMPORTS RPCError,      -- all D1 modules import this type.
        RPCDateTime    -- a particular RPC parameter type used here.
        FROM ECMABasicRPC-CallingSequences;
        -- clause 4.2 of this Standard
------------------------------------------------------------------------------------------------------
-- (The module now defines the specific D1 interface. This is an interface
-- declaration in the D1 notation defined in 6.2 and 6.3.)

-- Definition of the Application Service Element (ASE):
  dateTimeService APPLICATION-SERVICE-ELEMENT
                  CONSUMER INVOKES {getDateTime}
                  ::= {version 1}

-- Definition of the Operation identified in the ASE:
  getDateTime OPERATION
                  ARGUMENT  empty
                  RESULT    SEQUENCE {RPCStatusInfo, RPCDateTime}
                  ERRORS    {RPCError}
              ::= 1

-- This defines a procedure call with no input parameters. It has output
-- parameters (RPCOutput) consisting of status information (always defined for
-- all procedures) and a date/time parameter. The error type definition is the
-- same as for all procedures. The value notation at the end of the macro is the
-- integer encoding for this Operation name in the protocol (1, arbitrarily chosen).
```

-- Expected **RPCStatusInfo** contents in **RPCOutput** are defined in the following
-- table:

RPCStatus	RPCDiagnosticCode	RPCDiagnosticMessage
normal	0	Normal Result
warning	1	Clock unreachable, default time returned

-- The diagnostic information accompanying the **normal** status here is logically
-- redundant, but could be reassuring during system development. (The
-- diagnostics in **RPCStatusInfo** are, by definition, always optional).
--
-- In this example the **abnormal** status is not expected to occur, because there
-- are no **RPCInput** parameters and the functionality is very simple. Similarly, no
-- diagnostics are specified for the **error** case here (any such diagnostics would
-- be implementation-specific, and would probably only be relevant during server
-- program testing).

END -- Date Time Service definition.

F.3 Text File Service

TextFileService DEFINITIONS -- version 1-- ::=
BEGIN

-- This ASN.1 module is the D1 specification for a service which supports the
-- creation, deletion, reading and writing of text files. It is an example of how to
-- specify a service consisting of several Operations (procedures).
--
-- The interface to this service consists of the following operations (presented
-- here informally to explain their semantics and sequencing):

-- **resetFile** (File Name, File Handle)
-- opens file "File Name" for reading, returning a File Handle.
--
-- **rewriteFile** (File Name, File Handle)
-- opens file "File Name" for writing, returning File Handle; the file is created
-- if it did not previously exist.
--
-- **readLine** (File Handle, Destination Buffer)
-- reads the next line of information from a previously reset file.
--
-- **writeLine** (File Handle, Source Buffer)
-- writes the line at the current location of a previously rewritten file.

```
--
-- closeFile (File Handle)
--      closes a previously reset or rewritten file.
--
-- deleteFile (File Name)
--      deletes file "File Name".
-----------------------------------------------------------------------------------------------

-- The exports and imports defined below have the same structure as in all D1
-- specifications (e.g. they are copied from the previous example, except for the
-- different names and the importing of different RPC Parameter types).
-----------------------------------------------------------------------------------------------

-- To make the Application Service Element available to other ASN.1 modules:
   EXPORTS textFileServer


-- External definitions used:
   IMPORTS OPERATION
           FROM Remote-Operation-Notation
                {joint-iso-ccitt remoteOperations(4) notation(0)};

   IMPORTS APPLICATION-SERVICE-ELEMENT
           FROM Remote-Operation-Notation-extension
                {joint-iso-ccitt remoteOperations(4)
                 notation-extension(2)};

   IMPORTS RPCStatusInfo
           FROM ECMABasicRPC-ErrorManagement;

   IMPORTS RPCError, RPCCharacterString, RPCBinaryInteger,
           RPCVaryingStringDescriptor
           FROM ECMABasicRPC-CallingSequences;
-----------------------------------------------------------------------------------------------

-- Application Service Element (ASE) definition:
   textFileService APPLICATION-SERVICE-ELEMENT
                   CONSUMER INVOKES {resetFile, rewriteFile,
                   ReadLine, writeLine, closeFile, deleteFile}
                   ::= {version 1}
```

```
-- Operation definitions:

   closeFile     OPERATION
                      ARGUMENT     SEQUENCE {FileHandle}
                      RESULT       SEQUENCE {RPCStatusInfo}
                      ERRORS       {RPCError}
                 ::= 1

   deleteFile    OPERATION
                      ARGUMENT     SEQUENCE {FileName}
                      RESULT       SEQUENCE {RPCStatusInfo}
                      ERRORS       {RPCError}
                 ::= 2

   readLine      OPERATION
                      ARGUMENT     SEQUENCE {FileHandle,
                                             DestinationBufferDescriptor}
                      RESULT       SEQUENCE {RPCStatusInfo,DestinationBuffer}
                      ERRORS       {RPCError}
                 ::= 3

   resetFile      OPERATION
                       ARGUMENT     SEQUENCE {FileName}
                       RESULT       SEQUENCE {RPCStatusInfo, FileHandle}
                       ERRORS       {RPCError}
                  ::= 4

   rewriteFile    OPERATION
                       ARGUMENT     SEQUENCE {FileName}
                       RESULT       SEQUENCE {RPCStatusInfo, FileHandle}
                       ERRORS       {RPCError}
                  ::= 5

   writeLine      OPERATION
                       ARGUMENT     SEQUENCE {FileHandle, SourceBuffer}
                       RESULT       SEQUENCE {RPCStatusInfo}
                       ERRORS       {RPCError}
                  ::= 6

-- Type definitions:

   FileName                              ::= RPCCharacterString
   FileHandle                            ::= RPCBinaryInteger
   DestinationBufferDescriptor           ::= RPCVaryingStringDescriptor
   DestinationBuffer                     ::= RPCVaryingStringDescriptor
   SourceBuffer                          ::= RPCCharacterString
```

-- Expected content of **RPCStatusInfo** in **RPCOutput**:

```
-- ___________________________________________________________________ -
-- RPCStatus - RPCDiagnostic    - RPCDiagnosticMessage      - Operations -
--           - Code             -                           -            -
-- =================================================================== -
-- normal    - 0                - Normal Result             -   1-6      -
-- ___________________________________________________________________ -
-- error     - 1                - Invalid Name              -   2,4,5    -
--           -                  - Operation cannot be       -            -
--           -                  - performed on file named.  -            -
-- ___________________________________________________________________ -
-- error     - 2                - File Handle did not       -   1,3,6    -
--           -                  - correspond to opened file -            -
-- ___________________________________________________________________ -
-- warning   - 3                - Buffer too small          -   3        -
--           -                  - Line truncated.           -            -
-- ___________________________________________________________________ -
-- warning   - 4                - End of file reached, no   -   3        -
--           -                  - data in buffer.           -            -
-- ___________________________________________________________________ -
```

END -- Text File Service.

F.4 Eigenvalue Service

```
EigenvalueService DEFINITIONS -- version 1-- ::=
BEGIN
```

-- This ASN.1 module is the D1 specification for a service which computes
-- eigenvalues and eigenvectors. It is an example of how to define RPC
-- parameter lists that have floating point parameters.
--
-- The service supports one operation:

-- **calculateEigenvalues** (InputMatrix, Eigenvalues, Eigenvectors).
--   given an input matrix, the service computes its eigenvalues and
--   eigenvectors

```
  EXPORTS eigenvalueService
```

-- External definitions used:

```
  IMPORTS  OPERATION
           FROM Remote-Operation-Notation
                {joint-iso-ccitt remoteOperations(4) notation(0)};
```

```
IMPORTS  APPLICATION-SERVICE-ELEMENT
         FROM Remote-Operation-Notation-extension
              {joint-iso-ccitt remoteOperations(4)
               notation-extension(2)};


IMPORTS  RPCStatusInfo
         FROM ECMABasicRPC-ErrorManagement;


IMPORTS  RPCError, RPCArrayDescriptor
         FROM ECMABasicRPC-CallingSequences;
```

-- Application Service Element (ASE) definition:

```
eigenvalueService APPLICATION-SERVICE-ELEMENT
                  CONSUMER INVOKES {calculateEigenvalues}
                  ::= {version 1}
```

-- Operation definition:

```
calculateEigenvalues   OPERATION
                           ARGUMENT    SEQUENCE {EigenMatrix,
                                       EigenvectorDescriptor,
                                       EigenMatrixDescriptor}
                           RESULT      SEQUENCE {RPCStatusInfo,
                                       Eigenvector, EigenMatrix}
                           ERRORS      {RPCError}
                       ::= 1
```

-- Type definitions:

```
EigenMatrix                ::= RPCArrayDescriptor
Eigenvector                ::= RPCArrayDescriptor
EigenMatrixDescriptor      ::= RPCArrayDescriptor
EigenvectorDescriptor      ::= RPCArrayDescriptor
```

-- Expected content of **RPCStatusInfo** in **RPCOutput**:

```
-- ---------------------------------------------------------------------- -
-- RPCStatus  -       RPCDiagnosticCode      -   RPCDiagnosticMessage    -
-- ====================================================================== -
-- normal     - 0                            - Normal Result             -
-- ---------------------------------------------------------------------- -
-- warning    - 1                            - Ill conditioned matrix.   -
--            -                              - Possible underflow in     -
--            -                              - diagonalization.          -
-- ---------------------------------------------------------------------- -
-- error      - 2                            - Ill formed matrix.        -
--            -                              - Unable to diagonalize     -
--            -                              - matrix                    -
-- ---------------------------------------------------------------------- -
```

END -- Eigenvalue Service definitions.

F.5 Print Text File Service

PrintTextFileService DEFINITIONS -- version 1 -- ::=
BEGIN

-- This ASN.1 module is the D1 specification for a service which supports the
-- printing of text files. It is an example of how remote procedure callback is
-- specified, using LINKED child Operations.
--
-- This module defines the following operations:
--
-- **printTextFile** (FileName)
--   causes file "FileName" to be printed. The server prints the file by making
--   callback requests. Printing is completed when this operation returns.
--
-- **openFile** (Filename, FileHandle)
--   callback, opens file "Filename" for reading, returning a FileHandle.
--
-- **readBuffer** (FileHandle, DestinationBuffer, CharactersRead)
--   callback, reads the next buffer of information from the opened file,
--   and returns the number of characters read.
--
-- **closeFile** (FileHandle)
--   callback, closes the opened file.

```
-----------------------------------------------------------------------------------------------

   EXPORTS printTextFileService

-- External definitions used:

   IMPORTS  OPERATION
            FROM Remote-Operation-Notation
                 {joint-iso-ccitt remoteOperations(4) notation(0)};

   IMPORTS  APPLICATION-SERVICE-ELEMENT
            FROM Remote-Operation-Notation-extension
                 {joint-iso-ccitt remoteOperations(4)
                  notation-extension(2)};

   IMPORTS  RPCStatusInfo
            FROM ECMABasicRPC-ErrorManagement;

   IMPORTS  RPCError, RPCBinaryInteger, RPCVaryingStringDescriptor,
            RPCCharacterString
            FROM ECMABasicRPC-CallingSequences;
-----------------------------------------------------------------------------------------------

-- Application Service Element (ASE) definition:

   printTextFileService   APPLICATION-SERVICE-ELEMENT
                          CONSUMER INVOKES {printTextFile}
                          ::= {version 1}

-- Operation definitions:

   printTextFile   OPERATION
                       ARGUMENT    SEQUENCE {FileName}
                       RESULT      SEQUENCE {RPCStatusInfo}
                       ERRORS      {RPCError}
                       LINKED      {openFile, readBuffer, closeFile}
                   ::= 1

   openFile        OPERATION
                       ARGUMENT    SEQUENCE {FileName}
                       RESULT      SEQUENCE {RPCStatusInfo, FileHandle}
                       ERRORS      {RPCError}
                   ::= 2
```

```
readBuffer      OPERATION
                    ARGUMENT    SEQUENCE {FileHandle,
                                DestinationBufferDescriptor}
                    RESULT      SEQUENCE {RPCStatusInfo,
                                DestinationBuffer, CharactersRead}
                    ERRORS      {RPCError}
                ::= 3

closeFile       OPERATION
                    ARGUMENT    SEQUENCE {FileHandle}
                    RESULT      SEQUENCE {RPCStatusInfo}
                    ERRORS      {RPCError}
                ::= 4

-- Type definitions:
    FileName                                ::= RPCCharacterString
    FileHandle                              ::= RPCBinaryInteger
    DestinationBufferDescriptor             ::= RPCVaryingStringDescriptor
    DestinationBuffer                       ::= RPCVaryingStringDescriptor
    CharactersRead                          ::= RPCBinaryInteger

-- Expected content of RPCStatusInfo in RPCOutput:

-- ______________________________________________________________________ -
-- RPCStatus  - RPCDiagnosticCode  - RPCDiagnosticMessage    - Operations -
-- ====================================================================== -
-- normal     - 0                  - Normal Result           -    1-4     -
-- ______________________________________________________________________ -
-- warning    - 1                  - End of file reached     -    3       -
-- ______________________________________________________________________ -
-- error      - 2                  - Invalid filename        -    2       -
--            -                    - Operation cannot be     -            -
--            -                    - performed               -            -
-- ______________________________________________________________________ -
-- error      - 3                  - FileHandle did not      -    3,4     -
--            -                    - correspond to opened    -            -
--            -                    - file                    -            -
-- ______________________________________________________________________ -
```

END -- Print Text File Service definitions.