NISTIR 5277

Comparing Remote Procedure Calls

John Barkley

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

NIST

NISTIR 5277

Comparing Remote Procedure Calls

John Barkley

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

October 1993

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arati Prabhakar, Director

Abstract

Almost all computer systems are connected to a network supporting data communications. As a result, many techniques have evolved to support the development of applications which require processes on different systems to communicate and coordinate their activities. One such technique is remote procedure call (RPC). RPC is a mature method with several specifications and implementations. Among these are: Open Network Computing (ONC) RPC, Distributed Computing Environment (DCE) RPC, and the RPC specification from the International Organization for Standardization (ISO).

This report describes the RPC concept, how this concept is commonly implemented, and compares the features and capabilities of these three RPCs. The RPC language, semantics, and protocol of ONC RPC, DCE RPC, and ISO RPC are compared. Since ONC RPC and DCE RPC have implementations, the output of their RPC language compiler and the support provided by their runtime libraries are also compared.

Contents

List of Tables

List of Figures

1 Introduction

The Remote Procedure Call (RPC) concept is a simple and useful technique for developing applications where communication between cooperating processes on networked systems is required. RPC is a mature technique as evidenced by the existence of several RPC specifications and implementations[1].

This report describes and compares three significant RPCs:

- Open Network Computing (ONC) RPC from Sun Microsystems[SUN90][MS91].
- Distributed Computing Environment (DCE) RPC from the Open Software Foundation (OSF)[DCE91].
- The RPC specification from the International Organization for Standardization (ISO)[ISO91].

The criteria used for comparison is necessarily limited in scope because of the resources available to do the study. The author has experience in the development of applications using ONC RPC. While an implementation of DCE RPC was available, it was not possible to make use of the implementation because of limited resources. No implementation of ISO RPC is available. Nonetheless, the information presented in this report is significant for the task of choosing which RPC should be used in a given circumstance.

Section 1 is this introduction. Section 2 describes the RPC model, presents the RPC concept, and establishes some terminology. Section 3 discusses methods of implementing the RPC concept and presents further terminology used in the report. Sections 5, 6, and 7 describe features and capabilities of ONC RPC, DCE RPC, and ISO RPC respectively. Section 8 presents a table of comparison between the three RPCs which highlights significant similarities and differences.

2 The RPC Model

The RPC model describes how cooperating processes on different network nodes can communicate and coordinate activities. The paradigm of RPC is based on the concept of a procedure call in a programming language. The semantics of RPC are almost identical to the semantics of the traditional procedure call. The major difference is that while a normal procedure call takes place between procedures of a single process in the same memory space on a single system, RPC takes place between a client process on one system and a server process on another system where both the client system and the server system are connected to a network.

[1]Because of the nature of this report, it is necessary to mention vendors and commercial products. The presence or absence of a particular trade name product does not imply criticism or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available.

There are several representations of the original RPC model[BAD84]. Each of the references [SUN90], [DCE91], and [ISO91] presents a variation on the original suitable for the exposition of their RPCs. The RPC model of Figure 1 has been designed to illustrate the points of comparison used in this report.

Figure 1 illustrates the basic operation of RPC. A client application issues a normal procedure call to a *client stub*. The client stub receives arguments from the calling procedure and returns arguments to the calling procedure. An argument may instantiate an input parameter, an output parameter, or an input/output parameter. In the discussion of this Section, the term *input argument* refers to a parameter which may be either an input parameter or an input/output parameter, and the term *output argument* refers to either an output parameter or an input/output parameter.

The client stub converts the input arguments from the local data representation to a common data representation, creates a message containing the input arguments in their common data representation, and calls the client runtime, usually an object library of routines that supports the functioning of the client stub. The client runtime transmits the message with the input arguments to the server runtime which is usually an object library that supports the functioning of the server stub. The server runtime issues a call to the *server stub* which takes the input arguments from the message, converts them from the common data representation to the local data representation of the server, and calls the server application which does the processing.

When the server application has completed, it returns to the server stub the results of the processing in the output arguments. The server stub converts the output arguments from the data representation of the server to the common data representation for transmission on the network and encapsulates the output arguments into a message which is passed to the server runtime. The server runtime transmits the message to the client runtime which passes the message to the client stub. Finally, the client stub extracts the arguments from the message and returns them to the calling procedure in the required local data representation.

Like a normal procedure call, RPC is a synchronous operation, i.e., the client process is blocked until processing by the server is complete. This is not acceptable for many applications. As a consequence, the RPC model is enhanced to include the concept of a *lightweight process*. A lightweight process (also known as a *thread*) is an independent execution path within a normal process. A normal process can consist of several lightweight processes, each behaving like a normal process from the point of view of CPU use. However, all lightweight processes of the same process share the same address space. Thus, context switches between lightweight processes may be done more economically than context switches between normal processes.

In order to achieve asynchronous operation, a client application initiates an RPC call in a lightweight process and then proceeds with other processing. The application can recognize the completion of the RPC by some technique such as a status check or a software interrupt.

Note that, unlike the model presented here, some RPC models do not explicitly deal with the problem of systems which have different data representations. However, almost all implementations provide a mechanism to solve the problem of different data representations.

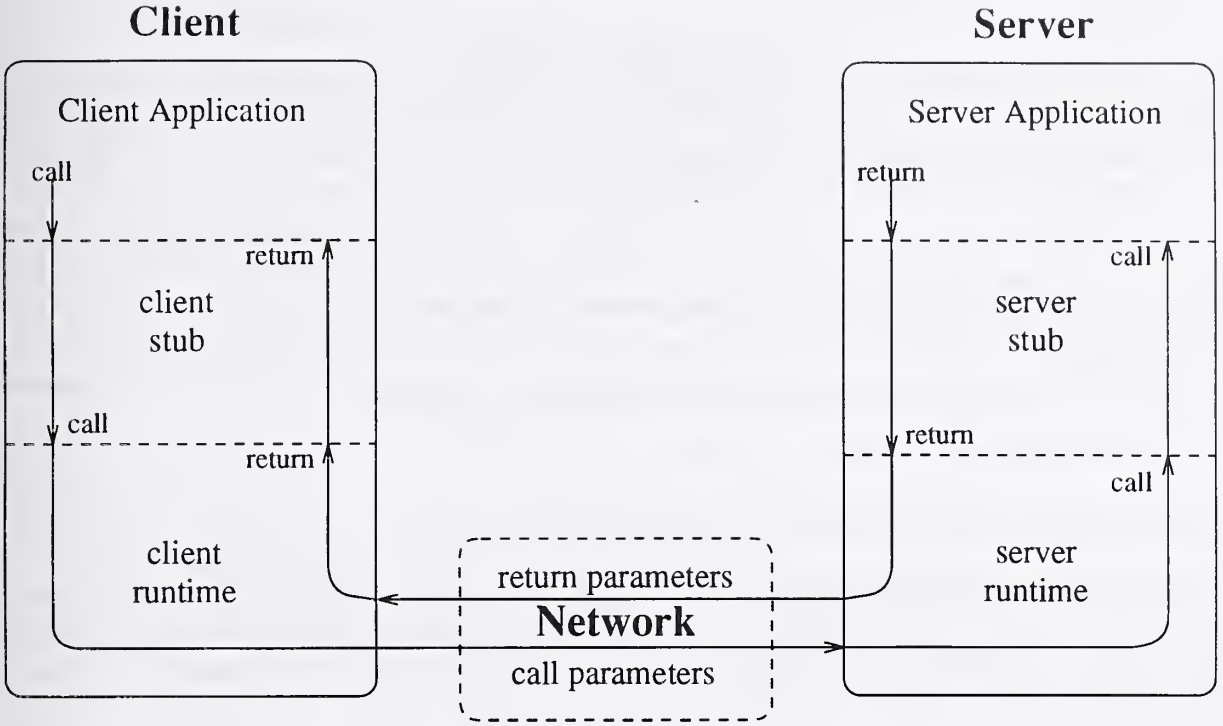

Figure 1: RPC Model

Usually, this mechanism is based on the use of a common data representation.

3 RPC Implementation

An implementation of the RPC model usually consists of at least three elements, a language compiler, a client runtime library, and a server runtime library. The language compiler produces suitable client and server stubs from a program written in an RPC language which is usually a non-procedural language providing the capability of declaring remote procedures and their parameters. In conjunction with the client and server applications, the client and server stubs are compiled by a procedure language compiler, such as C, producing object files which are linked to the client and server runtime libraries. This process produces an executable client and an executable server.

The client and server runtime libraries are called by the client and server stubs respectively. These object runtime libraries contain the routines for performing conversion between the local data representations and the common data representation, for creating the network message formats, and for the transmission of these messages between client and server according to user-specified protocols.

With such an implementation, the developer of an RPC application is required to produce the following:

```
long binop_add(a,b)
long a,b;
{
 return(a + b);
 }
```

Figure 2: The procedure *binop_add*

- The RPC language program for the RPC language compiler.
- The client application which calls the client stub.
- The server application which is called by the server stub.

In this report, a binary addition application is used as an example to illustrate these concepts. Consider a procedure named *binop_add* which adds two binary integer numbers. It has as input parameters two integers *a* and *b*. The return value of the procedure is the sum of *a* and *b*. In C, a call to such a procedure is:

c = binop_add(a,b);

In C, the procedure *binop_add* is implemented as shown in Figure 2. Following traditional practice, a developer wishing to use the procedure *binop_add* in an application simply calls the procedure and that procedure is linked into the application.

The goal of an RPC implementation is to achieve the same effect as the traditional method of using procedures except that the procedure *binop_add* is run on a separate system. Ideally, when *binop_add* is an RPC, the only additional task required of the developer is to produce declarations for *binop_add* and its parameters in the RPC language. The RPC language compiler generates the client stub which is called by the statement:

c = binop_add(a,b);

In addition, the RPC language compiler produces a server stub which calls the *binop_add* procedure in Figure 2. See Figures 3, 4, and 5 for the RPC language needed to define the *binop_add* example for ONC RPC, DCE RPC, and ISO RPC respectively.

3.1 Call Semantics

An RPC implementation may support more than one set of semantics for the RPC call. Which call semantics are used by a developer depends on the requirements of the application. The most common call semantics are idempotent and at-most-once.

3.1.1 Idempotent

Idempotent call semantics are for those RPC applications where no undesirable effects result from a given RPC call being processed on the server more than once. A client may issue a given RPC call more than once if the response to the first call is lost in transmission. An RPC application which maintains state information on a server may be adversely affected by multiple transmissions of the same RPC call. Idempotent call semantics are used by those RPC applications which do not maintain state information on the server (e.g., the *binop_add* application) or for those applications which do maintain state information but the server state is not corrupted by multiple client invocations of the same RPC call. Idempotent call semantics can be implemented in a very efficient manner.

3.1.2 At-most-once

At-most-once call semantics are for those RPC applications which require a guarantee that multiple invocations of the same RPC call by a client will not be processed on the server. Such applications usually maintain state information on the server and more than one invocation of the same RPC call must be detected in order to avoid corruption of the state information.

An example of such an application is inventory control. Consider several point-of-sale (POS) workstations and a server which maintains inventory records. Each POS workstation makes an RPC call to the server when an item is sold. The call causes a count of the number of items left in the inventory to be decremented on the server. If a call indicating that five of the item X was sold is processed on the server more than once, then the inventory record for item X will be in error. At-most-once call semantics provide a guarantee that this does not occur. However, at-most-once semantics exact higher overhead in terms of processing and network use and should only be used when necessary.

3.2 Variations on Call/Response Behavior

An RPC implementation may support different variations on the simple call/response behavior described in Section 2. These variations include broadcast RPC and an RPC for which no response to the call is required. In this report, an RPC which requires no response is referred to as a *no-response* RPC.

3.2.1 Broadcast

A broadcast RPC provides the capability for the application to make calls to more than one server with a single RPC invocation. For example, an inventory control application, such as the one described in Section 3.1.2, may be required to send point-of-sale information to several different servers maintaining inventory records.

3.2.2 No-response

The no-response RPC provides the capability for an RPC call to be made for which no response is returned. An example of such an application is a process that is monitoring some activity. The monitor process makes RPC calls to a server which maintains a log of the activity being monitored. The client need not get a response for each event logged.

4 Portability and Interoperability of RPC Applications

Among other things, information processing standards provide for portability and interoperability. The question can be raised with regard to which aspects of the RPC model and/or implementation are candidates for standardization. Based on the model illustrated in Figure 1, standard specifications could be developed for an RPC language, an RPC protocol, and/or client/server runtime libraries. Other aspects of RPC could also be considered for standardization. Discussion of which aspects of the RPC model and/or implementation should be standardized is beyond the scope of this report. For the purposes of the comparison of ONC RPC, DCE RPC, and ISO RPC, this report adopts the method of RPC standardization adopted by ISO in their RPC specification.

ISO RPC specifies both an RPC language and an RPC protocol. The specification of both an RPC language and an RPC protocol avoids the situation where an RPC language program may be ported to another system but the two systems may not interoperate because the client and server runtimes do not use the same RPC protocol. It is important to understand that ISO RPC does not specify a method of implementation. However, it is possible to describe how portability and interoperability for an RPC application is realized when a developer is using an RPC specification defined in the manner of ISO RPC and implemented in the usual technique as described in Section 3.

Portability of RPC applications is achieved at the source code level by means of the RPC language. Note that when an RPC specification defines only a language and a protocol, it is not necessary that the client and server stubs (i.e., the output of the RPC language compiler) be portable. The client and server stubs call routines in client and server runtime libraries which are likely specific to the system hosting the implementation of the RPC specification. On another host system, the RPC language compiler for that system generates client and server stubs suitable for the runtimes libraries of that system. Thus, the client and server stubs themselves may not be portable between host systems.

Consequently, in order for an RPC application to be portable when an RPC specification defines only a language and a protocol, it is necessary that the client and server stubs be *complete*. In this report, the terms *complete client stub* and *complete server stub* refer to the idea that the client stub and server stub generated by the RPC language compiler need not be modified by the RPC application developer in order to be integrated with the client and server applications. If an RPC application developer must modify the output of the RPC language compiler, then the portability of an RPC application is diminished.

Interoperability is achieved by means of the RPC protocol. Access to the RPC protocol is provided to the RPC application by the client and server runtime libraries. Because the RPC protocol used by an RPC application is part of the RPC standard specification, a client application on one system interoperates with a server on any other system.

5 ONC RPC

ONC RPC, sometimes referred to as Sun RPC, was one of the first commercial implementations of RPC. There are basically two implementations of ONC RPC, the original implementation and a transport independent implementation. The original implementation of ONC RPC is widely available on almost every system as part of the system's network software. In addition, source code for ONC RPC has been available over the Internet since 1988. This source code is readily usable on systems supporting Berkeley Unix libraries and has been modified for use on other systems.

The more recent implementation of ONC RPC is Transport Independent RPC (TI RPC)[MS91]. TI RPC is available as part of the Solaris operating system and is not as widely available outside of the Solaris environment as the original implementation of ONC RPC. The major difference between the original implementation and TI RPC is the ability of TI RPC to use different Transport Layer Protocols. TI RPC also provides somewhat more complete client and server stubs. In this report, discussion of ONC RPC always refers to the original implementation since this is the one which is so widely available. The differences between the original implementation and TI RPC do not significantly affect the major points of comparison as given in Section 8. Where a discussion refers to TI RPC, it is explicitly identified.

The success of ONC RPC is in some measure related to the widespread use of NFS which is implemented using ONC RPC. NFS server source code is also available on the Internet. More importantly, NFS has been implemented in many diverse environments, e.g., IBM MVS, DEC VMS, and Novell Netware.

5.1 Language and Semantics

ONC RPC supports both at-most-once and idempotent call semantics as well as broadcast RPC and no-response RPC (referred to as *batching* in ONC RPC). In ONC RPC, the language use to declare the procedures for client and server is called simply *RPC Language*[2]. ONC RPC supports three levels of authentication: none (the default), Unix user ID/group ID, and Secure RPC. Authentication using Unix user IDs and group IDs is well known as a very weak mechanism. Secure RPC is a very robust authentication mechanism which uses DES encrypted timestamps to authenticate.

[2]In this report, the term *RPC language* with the small "l" is used generically to refer to the input of an RPC compiler and the term *RPC Language* with a large "L" is used to refer specifically to the ONC RPC language.

```
program BINOP {
        version BINOP_VERS {

        long BINOP_ADD (struct input_args) =   1;

         } = 1;
} = 300030;

struct input_args {
        long a;
        long b;
};
```

Figure 3: Binary addition example in RPC Language

RPC Language provides support for almost all C language scalar and aggregate data types. In addition, it supports the data type **opaque** which permits untranslated data to pass between client and server.

With regard to procedure declaration, RPC Language is minimal. It only supports the declaration of procedures with one input parameter and one output parameter. However, this is a minor limitation. Input parameters may be grouped into a structure that becomes the single input parameter and output parameters may be grouped into a structure that becomes the single output parameter.

Client stubs in RPC implemetations typically provide for parameters which are both input and output. This can be accomplished in an ONC RPC client stub by copying an input/output parameter of the application procedure call to the input parameter structure of the RPC call. Upon return, the client stub copies the new value from the output parameter structure to the input/output parameter of the application procedure call. Unfortunately, the RPC Language compiler provides no support for this method of implementing input/output parameters.

Figure 3 shows the RPC language for defining the procedure and data for the binary addition example. The input parameters *a* and *b* have been grouped together into a structure *input_args* to provide the single input parameter to the RPC call. The application's call to the client stub in C would be:

```
c = binop_add(a,b);
```

The client stub prepares the *input_args* parameter from the *a* and *b* parameters of the applications call to the client stub.

Figure 3 also illustrates how the server procedure is identified in ONC RPC. The program *BINOP* is declared to be the number *300030* and the version number (*BINOP_VERS*) is declared to be *1*. The client stub invokes the procedure on the server by passing to the client

runtime services the name of the server, the program number *300030*, the version number *1*, the procedure name *BINOP_ADD*, and the input parameter *input_args*. Program numbers, version numbers, and procedure names neither require nor are supported by a name service. On the other hand, the name of the server requires and is supported by a name service.

5.2 Language Compiler

The RPC Language compiler is called *rpcgen*. It generates an include file, client stub, server stub, and a procedure to perform data representation translation of the input parameter and the output parameter to a common data representation for transmission over the network.

The client stub produced by *rpcgen* is incomplete. It is not ready to be used by the client application, i.e., the client stub generated must be enhanced to provide an application call of the form:

```
c = binop_add(a,b);
```

For the simplest cases of RPC applications, e.g., no authentication and idempotent call semantics, the procedure *callrpc()* is used to make the call from the client application and the client stub from *rpcgen* is unnecessary. In more complicated RPC applications, e.g., where the functionality of some kind of authentication and/or at-most-once call semantics is needed, the client stub needed requires at least twice the number of lines of code as is generated for the client stub by *rpcgen*. Again, this extra client stub code must be produced by the application developer.

On the other hand, the server stub produced by *rpcgen* is nearly complete for the majority of RPC applications. In the simplest applications, e.g., no authentication, the server stub is complete. For a more complicated application, e.g., some level of authentication, only minor modifications to the server stub must be made.

In addition to the files generated by *rpcgen* in the original implementation of ONC RPC, the *rpcgen* compiler of TI RPC produces a *makefile* (i.e., an input file to the *make* processor) which handles the generation of the client and server executables. Moreover the TI RPC *rpcgen* compiler produces a more complete client stub which is referred to as a "template." While this template provides considerable help, it must be modified in all cases by the developer of the RPC application.

5.3 Server Runtime Support

ONC RPC provides some support for concurrent execution of client calls on the server. The default sequence of events for server execution of a client call is generally as follows. When a client call arrives at the server, the server runtime support initiates execution of the server process for that call if the server process is not already processing a previous call. If the server process is already processing a previous call, then the new call is queued until the server process completes the previous call. This level of server runtime support is adequate for simple RPC applications.

However, for those applications which require more sophisticated support, ONC RPC provides the capability for server processes to be used in conjunction with the *inetd* daemon. This means that each time a client call arrives at the server, the *inetd* daemon initiates execution (using a *fork()/exec()* mechanism) of a server process for that call instead of queuing the call until a previous call completes.

5.4 Protocol

For reasons of efficiency, the protocol used for RPC calls in ONC RPC is a simple request/response protocol. The client sends a request packet to the server, the server processes the request, and returns a response packet to the client.

On its own, such a protocol is inadequate for supporting at-most-once call semantics. Even with the use of unique identification numbers on the request and response packets (called *transaction IDs* in ONC RPC), the server is unable to know whether the client received the response packet unless there is some acknowledgement from the client.

ONC RPC solves this problem by relying on the transport layer protocol to provide the call semantics, i.e., UDP for idempotent calls and TCP for at-most-once calls. In those applications where idempotent call semantics are sufficient, this approach minimizes the number of packets on the network and the amount of packet processing by client and server. However, in those applications where at-most-once call semantics are required, the use of TCP usually generates more packets and may increase the amount of packet processing as compared to a more robust RPC protocol. Such a more robust RPC protocol can support at-most-once call semantics over a connectionless transport protocol such as UDP.

The original implementation of ONC RPC only supported UDP and TCP transport protocols. The more recent implementation of ONC RPC, TI RPC, provides support for a larger group of transport protocols including OSI Transport.

In order to solve the problem of different data representations between different systems, ONC RPC uses a standard data representation for both scaler and aggregate data types. ONC RPC calls this standard data representation *External Data Representation* (XDR)[SUN90, Chapter 5]. The client input parameter is converted to XDR representation. The server converts the XDR representation of the client input parameter to its local representation. Having produced the results of the client request, the server prepares an XDR representation of the output parameter. The client converts the XDR representation of the output parameter to its local representation.

6 DCE RPC

DCE RPC is not nearly as widely and easily available as ONC RPC. Although support for DCE, of which DCE RPC is a part, is claimed by almost all vendors, DCE RPC is a future product for some vendors. For other vendors, DCE RPC is an option over and above the normal networking software which usually includes ONC RPC.

6.1 Language and Semantics

DCE RPC is rich in both language and semantic features. DCE RPC supports both at-most-once and idempotent call semantics as well as broadcast RPC and no-response RPC (referred to as *maybe* in DCE RPC). DCE RPC supports several authentication mechanisms. The default is Kerberos Version 5 (called *DCE shared-secret*). Any other authentication mechanism including no authentication must be specifically requested by the client. The language of DCE RPC is called *Interface Definition Language* (IDL). DCE RPC is descended from Network Computing System (NCS) RPC developed by Apollo and provides conversion tools to go from NIDL (the NCS language compiler) to IDL.

In its procedure definitions, IDL supports procedure declarations with input (**in**), output (**out**), and input/output (**in/out**) parameters. IDL also generally supports all of the C language data types, the **handle_t** data type for untranslated data, plus some additional ones. For example, IDL supports a **pipe** data type. It is sometimes desirable in an RPC application to be able to transfer large amounts of data between client and server. The traditional means of accomplishing this has been to establish communications channels between client and server as a result of the RPC call. With ONC RPC, there is no support within RPC Language for this kind of application. Such communications channels must be programmed by hand.

For most such applications, the DCE RPC data type **pipe** provides the capability needed. A **pipe** can be an input parameter (a client to server channel), an output parameter (a server to client channel), or and input/output parameter (a two-way channel between client and server). The **pipe** permits the transfer of large amounts of typed data between client and server. Because the data that passes over a **pipe** parameter is typed, DCE RPC, by default, provides the necessary translation between the local representations of the typed data and the common data representation for transmission over the network. With ONC RPC, such translation must be programmed using a data translation procedure. The RPC Language compiler *rpcgen* may be used to generate such a data translation procedure. Note that a DCE RPC procedure that has a **pipe** parameter cannot be idempotent.

Figure 4 shows the IDL for the binary addition example. The call to the client stub in C from the application program is:

```
binop_add(h, a, b, &c);
```

The argument *h* is a handle which locates state information that is kept on the client. The sum is returned by means of a pointer to *c*.

The *uuid*, *version*, and *endpoint* provide a unique identification of the interface on a server. Normal use of DCE RPC requires a name service. However, for development and debugging purposes, the name service need not be used.

6.2 Language Compiler

The IDL compiler produces complete client and server stubs where the default call semantics are at-most-once and the default authentication mechanism is Kerberos. The use of other

```
/*
 * (c) Copyright 1990, 1991, 1992 OPEN SOFTWARE FOUNDATION, INC.
 * ALL RIGHTS RESERVED
 */
/*
 * OSF DCE Version 1.0, UPDATE 1.0.1
 */
/*
*/
[uuid(f9f6be80-2ba7-11c9-89fd-08002b13d56d),
 version(0),
 endpoint("ncadg_ip_udp:[6677]", "dds:[19]")]
interface binopwk
{

    [idempotent] void binopwk_add
    (
        [in] handle_t h,
        [in] long a,
        [in] long b,
        [out] long *c
    );
}
```

Figure 4: Binary addition example in IDL

authentication mechanisms requires minor modifications to the client and server stubs.

6.3 Server Runtime Support

By default, DCE RPC provides concurrent execution of client calls on the server. Each time a client call arrives at the server, the server creates a new thread for the server process to handle the client call. A client call is queued only if there are no resources on the server system for a new thread to be initiated. In DCE RPC, the use of threads is an integral part of the DCE RPC implementation. With ONC RPC, threads may be used by an RPC application whenever threads are supported on the system hosting the ONC RPC implementation. Threads are not an integral part of ONC RPC.

6.4 Protocol

The protocol used for RPC calls in DCE RPC depends on which call semantics have been specified in IDL. For idempotent semantics, a simple request/response protocol is used. For at-most-once semantics (the default), the protocol is:

1. Client sends request packet to server.
2. Server sends response packet to client which acknowledges to the client the receipt of the request packet by the server.
3. Client sends acknowledgement to server indicating receipt of the response packet.

This protocol, along with the use of unique identification numbers (called *invoke IDs* in DCE RPC) with the request and response packets is the means by which DCE RPC supports at-most-once call semantics. This allows DCE RPC to support at-most-once semantics over a connectionless transport protocol such as UDP.

Like ONC RPC, DCE RPC solves the problem of different data representations on different systems by translating local data representations into a common data representation[3] called *Network Data Representation* (NDR)[DCE91, Section 10.5]. In addition, the protocol supports the negotiation of data representations but currently, NDR is the only data representation used.

The use of NDR is a departure from the technique used to solve the problem of different data representations by NCS RPC from which DCE RPC is derived. With NCS RPC, it is the sole responsibility of the server to interpret and translate between local client data representations and local server data representations. In NCS RPC, there is no data representation (such as NDR) which is neutral to all systems.

7 ISO RPC

ISO RPC (ISO/IEC CD 11578-1, SC 21 N 6561) is in a committee draft stage in the ISO standard development process. The reference [ISO91] used in this report is the draft for a CD letter ballot to be returned by March 20, 1992. It consists of four parts:

1. Model - defines an "RPC Interaction Model" and an "RPC Communications Model."
2. Interface Definition Notation (IDN) - defines the language for specifying RPC procedure and data declarations.
3. Service Definition - defines the services for "a Basic RPC User Application-service-object (ASO), Signature Application-service-element (ASE), a Basic RPC ASO, and the Basic RPC ASO-association Object."
4. Protocol - defines RPC protocol.

There are no commercial implementations of ISO RPC available.

[3] referred to as *transfer syntax* in DCE RPC

```
interface BINOP
        begin
           procedure BINOP_ADD(
                in a:integer,
                in b:integer
                )
              returns(c:integer)
              raises( OVERFLOW() )
        end
```

Figure 5: Binary addition example in IDN

7.1 Language and Semantics

Unlike ONC RPC and DCE RPC, ISO RPC specifies that all RPC calls have at-most-once call semantics. There is no mention in ISO RPC concerning how authentication is to take place.

Like DCE RPC IDL, ISO RPC IDN supports input, output, and input/output parameters. Unlike ONC RPC and DCE RPC, IDN supports the declaration of *exceptions*, i.e., procedure error returns which can have their own parameters. The number of data types supported is less than the number supported in either ONC RPC or DCE RPC. For example, RPC Language and IDL support the aggregate data type **union**.

Figure 5 shows how the binary addition example might look in IDN. Note that in a real implementation of IDN, the interface name (*BINOP*), the procedure name (*BINOP_ADD*) and the exception name (*OVERFLOW*) may have conventions beyond the syntax specified in the ISO RPC specification in the same manner as RPC Language and IDL. The example also shows how an exception condition can be defined in IDN. In this case, the exception condition indicates an overflow from the addition.

There are many unresolved questions in the ISO RPC specification. Annex A of the part of the specification which defines the model (ISO/IEC CD 11578-1) contains a list of these questions for "further study."

7.2 Protocol

ISO RPC specifies the use of other existing or emerging ISO Application Layer protocols. The ISO RPC protocol requires the use of Remote Operation Service Element (ROSE) and Association Control Service Element (ACSE). It is not clear from the specification how these protocol elements combine to support at-most-once call semantics. Since the ISO RPC protocol is an Application Layer protocol, data representation differences between client and server are handled by the ISO Presentation Layer protocol.

The concept that the ISO RPC protocol resides at the Application Layer is highly con-

	ONC	DCE	ISO
commercial availability	almost everywhere as part of network software	future product or option to network software	none
idempotent call	yes	yes	no
at-most-once call	yes	yes	yes
broadcast RPC	yes	yes	no
no-response RPC	yes	yes	no
default authentication	no authentication	Kerberos Version 5	none specified
optional authentication	Unix UID/GID, Secure RPC	no authentication and others	none specified
parameter direction attribute	one input, one output	input, output, input/output	input, output, input/output
completeness of stubs	poor for client, adequate for server	good for both client and server	not applicable
concurrent server call processing	default is none, optionally *inetd*	always by threads	not applicable
at-most-once implementation	by means of TCP as transport layer	by means of RPC protocol	not clearly specified
data representation	XDR	NDR	ASN.1

Table 1: Comparison of ONC, DCE, and ISO RPCs

troversial. Those in favor of the RPC protocol residing at the Application Layer argue that it is properly located because, in order to solve the problem of different data representations, the Presentation Layer is required. This point of view is consistent with the concepts in the ISO Seven Layer Model.

Those who argue against the RPC protocol residing at the Application Layer point out that many RPC applications are required to be very efficient. In their opinion, this is not possible with an RPC protocol at the Application Layer. The RPC protocol should reside in the Session Layer with simply a common data encoding rather than a whole protocol layer (i.e., the Presentation Layer) to provide for a common data representation.

8 Summary

Table 1 summarizes some of the important characteristics of ONC RPC, DCE RPC, and ISO RPC discussed in the body of the report. As indicated by the table, DCE RPC has the advantage over the others from the point of view of overall capability. DCE RPC supports at-most-once semantics in an efficient manner and permits the optional use of idempotent

semantics in an efficient manner. DCE RPC provides a robust authentication mechanism and better support for the developer of RPC applications in that the IDL compiler produces complete stubs and the server runtime defaults to concurrent execution of calls. Even though DCE RPC is not as widely available as ONC RPC, ONC RPC most likely comes as part of the network software that supports DCE RPC. Consequently, ONC RPC is available at no extra cost in almost all cases.

ONC RPC has an advantage in that it is widely available. However, it does not provide the most efficient means possible for supporting at-most-once semantics. Moreover, the client stub generated by the RPC Language compiler is incomplete and requires hands-on augmentation by the developer.

ISO RPC has no current commercial implementation. Therefore, it is difficult to make a fair comparison. The current state of the specification leaves some characteristics of ISO RPC as yet unspecified or not completely specified. Thus, it is unlikely that a commercial implementation will become available soon.

References

[BAD84] Nelson B. J. Birrell A. D. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[DCE91] OSF DCE 1.0 Application Development Guide. Technical report, Open Software Foundation, December 1991.

[ISO91] ISO Remote Procedure Call Specification. ISO/IEC CD 11578 N6561, ISO/IEC, November 1991.

[MS91] Chuck McManis and Vipin Samar. Solaris ONC: Design and Implementation of Transport-Independent RPC. Solaris 2.0 White Papers, SunSoft, 1991.

[SUN90] Sun Microsystems Inc. *Network Programming Guide*, Revision A March 27 1990.