Remote Procedure Call: Technology, Standardization and OSF's Distributed Computing Environment

Dietmar Fauth
Siemens Nixdorf Informationssysteme AG

Chikong Shue
Open Software Foundation, Inc.

Contents

1. Introduction

2. The RPC Programming Paradigm

   2.1 Message I/O versus Remote Procedure Call
   2.2 Challenges of Remoteness in Procedure Calls
   2.2.1 Location and Binding
   2.2.2 Procedural Information Transfer
   2.2.3 Error Handling
   2.2.4 Ease of Use
   2.2.5 Security
   2.2.6 Concurrency
   2.2.7 Management
   2.3 Model of a Generic RPC

3. OSI RPC Standardization

   3.1 OSI RPC Modelling
   3.2 OSI RPC Requirements
   3.3 OSI RPC Service Primitives
   3.4 OSI RPC Protocols
   3.5 Interface Definition Notation

4. Architecture of the OSF™ RPC

   4.1 Interface Definition
   4.2 Run-time System
   4.2.1 Communication Service
   4.2.2 Naming Service
   4.2.3 Management Service

5. RPC as Used in the OSF™ Distributed Computing Environment

6. Conclusion

Copyright 1991, 1992. Open Software Foundation, Inc. All Rights Reserved.

Remote Procedure Call: Technology, Standardization, and OSF's Distributed Computing Environment

1. Introduction

The widespread use of cost-effective workstations, personal computers, and specialized servers interconnected via ubiquitous local-area and high-performance wide-area networks has created a demand for distributed applications. In order to build distributed applications, developers need an easy-to-use programming paradigm that will allow them to take advantage of the distributed computing architecture. That paradigm is the remote procedure call, or RPC.

Using RPC technology and following the client/server model, developers can easily create distributed applications for use in heterogeneous computing environments. The RPC mechanism partitions the various tasks an application requires into different procedure modules that can be performed on different servers. This technology offers several benefits: an easy-to-use programming model, the balanced, distributed use of computing resources, high availability of services through replication, and the ability of applications to run across diverse software and hardware platforms.

The RPC paradigm implements the model of the simple procedure call within the client/server architecture. It hides from the programmer various network complexities such as the means for locating and accessing the hosts on which required services are running. As a result, RPC is a general-purpose tool that can be used universally, for applications in such diverse fields as office services, factory automation, and distributed file systems.

RPC technology currently has wide support in the industry. This paper provides an overview of the benefits of using RPC technology to develop distributed applications. In addition, it provides an RPC architecture as well an overview of an RPC implementation.

In the chapter following this overview, we recapitulate the familiar model of procedure calls as they are used with every higher programming language. Transferring this model to the distributed environment reveals new aspects that are inherent to the newly introduced concept of remoteness. RPC must cope with these differences. In order to overcome the difficulties associated with distribution, RPC necessarily must provide certain mandatory features. The requirements that will be derived this way can be used to architect a generic RPC.

In the subsequent chapter, the ISO working version of RPC is presented as it exists today. Thereafter, a case study illustrates how a state-of-the-art RPC system is designed and how it is being utilized by various components in building the Distributed Computing Environment (DCE) - a distributed software platform offered by Open Software Foundation (OSF™).

2. The RPC Programming Paradigm

2.1 Message I/O versus Remote Procedure Call

The traditional programming paradigm between distributed applications (also known as communication service users) and a communication service provider is based on message I/Os (input/output) as depicted in Figure 1A. There are three parties involved in this model: 1) *a client application* (e.g., a file client), 2) *a server application* (e.g., a file server) and 3) *a communication service provider* (e.g.. a transport service or a presentation service provider). In order to achieve a request-response exchange between a client and a server application, application developers first have to set up a service provider (say, X) connection between the client and the server, then interact with the X service provider using service primitives such as X-DATA.request, X-DATA.indication, X-DATA.response and X-DATA.confirm. X-DATA.request

and X-DATA.response are usually I/O calls whereas X-DATA.indication and X-DATA.confirm are usually I/O interrupts. Programmers have to manipulate fields in the request messages (e.g., read, write, search) and fields in the corresponding response messages (e.g. success/failure, results). These message I/O interfaces are not only hard to use and maintain but also difficult for developing distributed applications in a structured way.

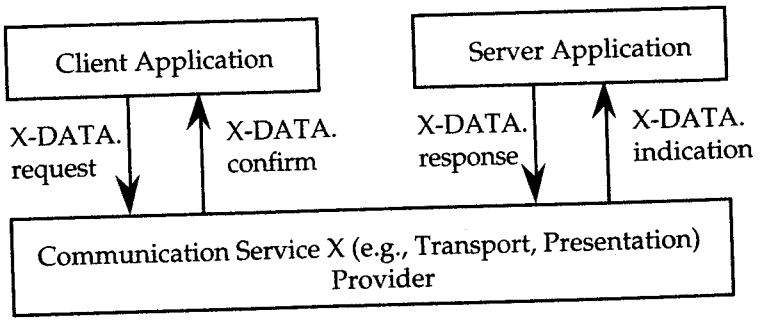

Figure 1A Message I/O Paradigm

RPC provides a higher level of abstraction in modelling distributed applications as shown in Figure 1B. In this model, a client sending a message to a server and getting a response is like a program calling a procedure and getting a return. There are only two parties involved in this model: 1) *a client application* making remote procedure calls and 2) *a server application* executing the calls and returning results. The complicated interactions with the service provider like connection setup/teardown, and message I/Os and interrupts are completely hidden from the RPC service users. This provides a much simplified view for application developers because making remote procedure calls resembles local procedure or function calls in higher programming languages, in which one procedure calls another to fulfill a certain task. A calling program passes a number of parameters to the called procedure or function, which after executing the desired operation, returns the result in the form of output parameters and/or a return value (functional result) to the calling program. If for some reason the operation could not be executed or completed, an error code is returned instead. All this happens in a synchronous manner (i.e., while the called procedure is executing, the calling procedure is blocked until the result is returned).

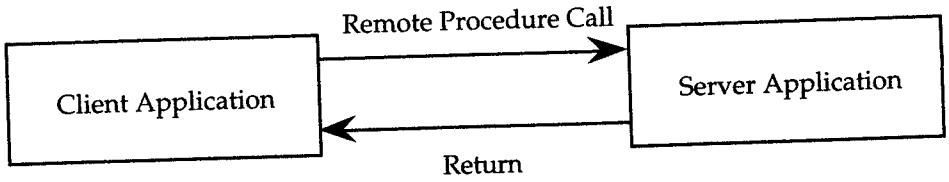

Figure 1B Remote Procedure Call Paradigm

Even in the local case just described, a client-server paradigm applies where the calling function can be identified with a client whereas the called procedure plays the part of the server. The actual services delivered in the local environment are typically more specific and subject to local constraints; thus the local case can be considered as a degenerate version of the distributed case.

As a first try RPC can be modeled along the same lines; there is a calling procedure, the client, which passes parameters to a called procedure, the server. The client then waits synchronously for the server to finish its task and deliver the results. In this case, however, the only difference being now that clients and servers are running in different processes and typically, on different machines. In principle, the physical separation of the communicating programs can be completely transparent to the logic of the application.

However, different degrees of transparency are sometimes desired to allow flexibility and optimization. For example, a user of a distributed printing service may want to specify where a document is to be printed. Thus, besides a completely transparent mode, a means of indicating where the operation is preferred to be performed is desirable.

2.2 Challenges of Remoteness in Procedure Calls

The comfort of having an entire network's resources at one's disposal via remote procedure calls has its price; one must cope with the challenges resulting from the physical separation of the client and the server when an application is distributed over a network. Resolving these difficulties will lead to the architecture of a generic RPC.

2.2.1 Location and Binding

A basic problem exists when the calling and called procedures do not reside on the same machine: the client must find an appropriate server somewhere in the network. There are at least three possible solutions for resolving the addressing issues. First, location information on known servers, which consists of transport address and port address, could be inserted into the code at coding time; second, environment variables could be used for evaluation at compile time; or the identification of the RPC interface - the externally visible characteristics of a set of procedures the server offers - could be used to map to server addresses at run-time. The last approach is clearly the most flexible because changes in the configuration of the distributed system remain hidden from the application.

A trading mechanism must exist in order for this scheme to work. A server must register with a naming service declaring its readiness to accept requests via a particular interface. The registration operation invoked by the server is also known as "exporting" the interface. A client application will query the naming service with a name, object identifier or attributes as an input that specifies the interface. The addressing information can be returned by the naming service in a data structure that may be completely opaque to the requesting client. If the client wants to have influence on the site of the execution of the operation, it must check this structure and may request new ones until an appropriate one is found. The locating operation invoked by the client is also known as "importing" the interface.

Once a client obtains the location of a server, it must establish a binding to the server before any remote procedure can be called. The client may utilize different binding techniques or policies (e.g., bind before the first call, bind at the time of the first call, rebind for each call, automatic rebind to another available server when the server crashes) in order to meet its specific application requirements.

Thus, the first features required to accommodate the remoteness of the procedures are the presence of a naming service, a trading mechanism, and a binding service.

2.2.2 Procedural Information Transfer

After a client determines the server's location, it must determine how to communicate with the server. In a world of diverse, incompatible transport mechanisms, a client must find out which protocol stack it needs to use to reach the server. Thus, if more than one transport is to be supported, this must be stored in the naming service for each server along with the addressing information. The same holds true if a server supports more than one RPC protocol.

In addition, a server could use the same naming service to store information describing network data representations (i.e. transfer syntaxes) it understands. For greater flexibility, OSI has defined a mechanism of presentation context negotiation whereby the semantics of the data exchanged as well as their wire representation are agreed on between the communicating partners. Due to the nature of heterogeneity, an RPC system must be able to convert between the transfer syntaxes and the local syntax when necessary.

Parameter passing in remote procedures can only be done in the "pass-by-value (copy/restore)" scheme. It is achieved by assembling and encoding (i.e., marshalling) parameter values together as a data stream prior to transmission; and by disassembling and decoding (i.e., unmarshalling) the parameter data stream into parameter values after the data stream arrivers. RPC has to resolve the mapping of parameter values between two different address spaces, i.e. the client address space and the server address space. Take a "pointer" datatype as an example, an RPC system should be able to transfer graphs, trees, or linked lists connected by pointers between the client address space and the server address space and to reflect the value as well as topology changes from the server address space to the client address space. Moreover, it also must handle the aliasing case, in which a node contained in a parameter can be pointed to by other parameters. Although all these pointer handling mentioned above are taken for granted in local procedures, they present real challenges in remote procedures.

In a distributed system, application data needs to be transferred from a client to a server and vice versa. In a connection-oriented environment, for example, connections have to be established and released later with data transferred in between. If a connectionless transport is employed, there must be a mechanism to monitor the data exchange to ensure the correct order of incoming packets or test the liveness of the peer process.

In either case, these tasks require a run-time system (RTS) to set up the environment for communication. The aforementioned problem of different data representations on different hosts is one sub-task addressed by a dedicated part of the RTS.

2.2.3 Error Handling

In a distributed environment, errors that are not possible in a local environment may occur. In the local case, a severe error in the called procedure (e.g., divide-by-zero) leads to a failure of the entire process, caller included. If the error is not fatal, the called procedure returns with an appropriate error message.

In the distributed case, communications and environmental errors (communication link down, server crashes, etc.) may occur during the execution of a remote procedure call. It is essential that these errors report back to the client that is awaiting the result of the call. There are several different ways to report errors mentioned above. For example, they may be reported via a special parameter included in the call, via a global variable, via a system call or via exception handling mechanisms. In any case, the client must check these errors to determine whether the remote procedure has been executed successfully. A way to reinstall the surviving party to a definite state after a fault occurred on the other side is also desirable. This is also true if, in the connection- oriented environment, a connection after a breakdown cannot be recovered in a specified period of time. Thus certain features such as clean-up routines or cancellation operations must be part of of a well-designed RTS.

2.2.4 Ease of Use

A local procedure call paradigm is familiar to every programmer. It is desirable to maintain this simplicity in a distributed environment as much as possible. However, as demonstrated in this chapter so far, there are many non-trivial steps to be taken when a remote procedure call is being made.

Thus a tool which relieves a programmer of the work described up to now must be provided. Such a tool exists for all useful RPCs. It is presented to a programmer in the form of an interface definition notation (IDN). An IDN is used to specify procedures that are designed to run remotely. An IDN specification serves as an input to an IDN compiler which automatically generates stubs for both client and server programs; the stubs invoke the RTS calls necessary for transferring and converting data to the desired representation. On the client's side, the client stubs which are linked together with the client application can be viewed as the surrogate entry points for the remote procedures residing on the server. Likewise, the same statement is true for the server stubs where the role of client and server are reversed.

The IDN serves to specify a set of remote procedures for interfaces. The specification of remote procedures comprises the identification of procedures and their parameters. Well-designed IDNs allow the specification of different call semantics for the remote procedures (e.g., at-most-once, idempotent, callback). Moreover, they provide an extensive set of data types and attributes for the parameters such as simple types (e.g., integer, character) and constructed types (e.g., structure, union, array, pointer).

2.2.5 Security

The distributed computing environment must guarantee that access to services and objects be limited to authorized clients. Each client also must be able to select a specific server and be assured that only this server gets access to its sensitive data during callbacks.

This kind of security requires the authentication of either side so that each peer application can be certain of its partner's identity. When both client and server have verified each other's identity, a method of authorization must be in place to regulate the access of clients to server objects.

Further important security concerns are the data confidentiality (privacy) and data integrity of parameters passed back and forth during remote procedure calls. The sender of remote procedure calls wants to be sure that his/her messages are neither read nor distorted by a third party.

Consequently, a secure RPC requires an authentication service as well as an encryption technique on which all these different aspect of security rely. The RPC protocol must be able to support these security services. Moreover, the RPC system should allow an application to decide at run time which degree of security is appropriate.

2.2.6 Concurrency

In a distributed system, multiple clients may request a single server to perform operations at the same time. Thus a server can either put the operation requests in a queue and process them one at a time according to some policy, or otherwise allow for concurrent execution of operations. The latter approach is both more efficient and more equitable because it avoids uncertain waiting periods for operations that were initiated by unrelated clients.

On the other hand, a client application might have to utilize several services simultaneously. Instead of waiting for the end of one operation before requesting another, passing subsequent operations to various qualified servers asynchronously might be more efficient. In this way, the concept of concurrency appears on the client side as well.

A convenient approach that frees application programmers from dealing with the logics of asynchronous calls and instead employs the familiar paradigm of local synchronous function calls is provided by lightweight processes or threads. An obvious way to use threads in a server is to have a listener thread that spawns a new thread for every incoming procedure call. Likewise, a client main thread creates a new thread for each outgoing procedure call, waits for each thread to terminate and receives the result of each call. The client also can alert a thread in order to cancel a outstanding call if desired.

The inherent nature of concurrency in distributed environment provides a new challenge as well as opportunity for RPC systems.

2.2.7 Management

If a distributed system is to run smoothly, there must be a way to manage this system remotely because it is not feasible for a system manager to travel from site to site in order to check on servers. Some management functions are independent of the kind of service a specific server offers. Such generic management functions are, for example, those that inquire whether a server is still working or which interfaces it has currently registered. It is the RPC run-time system's job to provide an application with that kind of information.

2.3 Model of a Generic RPC

Extending the simple model of a function call to a distributed environment, as was done in the preceding chapter, leads to an RPC architecture as shown in Figure 2. Actually, it is not just the structure of RPC that could be derived in this way, but rather the necessary ingredients that make up the distributed system. This model identifies new components necessitated by the remoteness in which the assumption that called and calling procedure run in the same process is dropped. The fact that such a profound result follows from what appears at first sight to be a simple idea emphasizes the central role of RPC in a distributed environment.

Figure 2 summarizes the structure of an application that employs the RPC model; further, it includes a listing of the functionalities an RPC must possess.

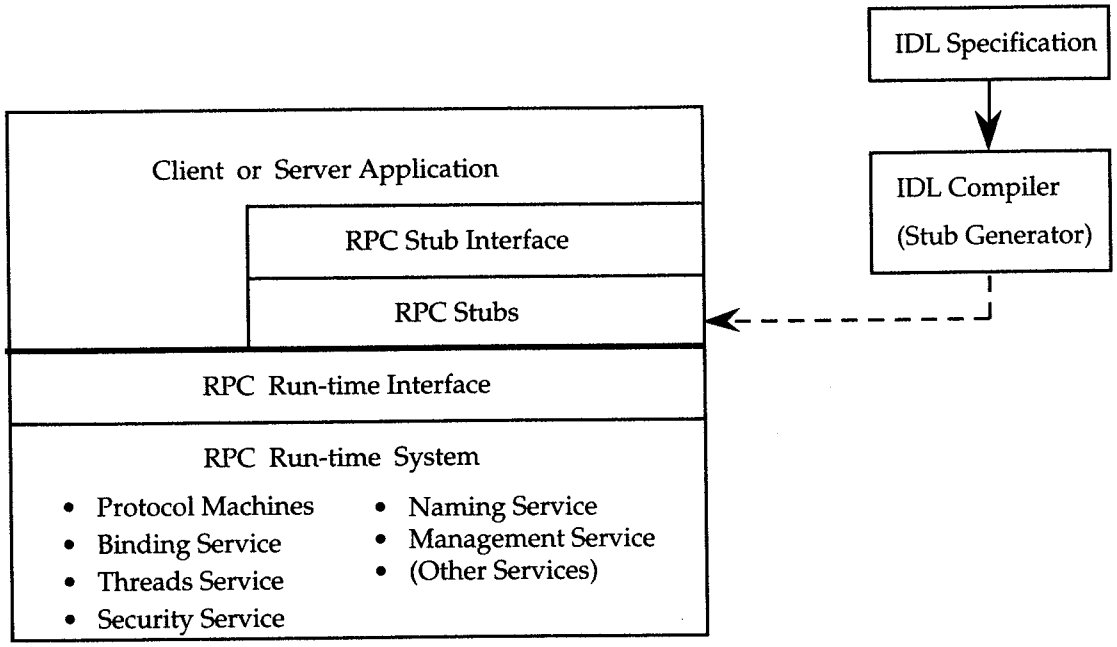

Figure 2 Generic RPC Architecture

3. OSI RPC Standardization

Since the mid 1980s, many proprietary RPC facilities have become available on various hardware and software platforms. They are gaining popularity and are recognized as powerful tools in building distributed applications. In order to ensure the future interoperability of RPC-based distributed applications, ISO/IEC JTC 1/SC 21 started a new project on RPC standardization at the end of 1989. The first working draft for the OSI RPC standard was produced at the first official RPC Rapporteurs Group meeting held in May, 1990, in Seoul, Korea. The working draft evolved into a second version during the Group's Interim meeting held in November, 1990, in Valbonne, France.

The interoperability of RPC-based applications has a very significant consequence. It will enable the designer of a server application (e.g., on a Mainframe) to publish the server interface in a formal, non-ambiguous specification in a standard interface definition notation (which can be considered a network application contract), and allow client applications (e.g., on a PC) to be developed independently by following the published interface specifications. The OSI RPC standard protocols will allow the client applications and server applications to interoperate in heterogeneous environments.

A brief up-to-date summary of the current working draft is provided below.

3.1 OSI RPC Modelling

The OSI RPC standard provides both a computational model and an engineering model. The computational model describes the type of program interaction to be supported by the standard. The RPC engineering model describes how the RPC computational model can be realized. The concepts in the RPC computational model are nearly complete and will be described further. The current draft on the RPC engineering model is not stabilized yet. In this paper, we'll attempt to use the OSF RPC architecture as a real engineering solution with an attempt to realize the OSI RPC computational model.

In the RPC computational model, a set of user-defined remote procedures is arranged to create an interface. All interfaces are typed, and the interface type is defined by the externally visible characteristics of the set of procedures. An RPC server offering the set of procedures characterized by a particular interface type supports an interface instance of that type.

The specification of an interface type is termed an interface definition. Uses of the definition include documenting and exchanging information about or registering a particular interface type. Interface definitions are written using the interface definition notation (IDN).

An RPC binding is a relationship between an RPC client and an RPC server, using an interface. Individual bindings are strictly one-to-one relationships. However, a single client may be involved in multiple bindings with one or more servers, and a single server may be involved in multiple bindings with one or more clients. Multiple bindings may exist between a particular client and a particular server, using one or more interfaces.

Establishing the binding requires that the RPC client provide information sufficient to identify the desired interface. A successful binding must be established before any remote procedures can be called. OSI RPC supports various binding policies, e.g., bind at the time before the first call, bind at the time of the first call. Binding policies may be customized to meet the specific needs for either client or server environments; therefore, they are not specified in the interface definition.

Orderly release of active bindings can be initiated by the RPC client only.

The interaction semantics of OSI RPC emulates the procedure call/return semantics of local procedure calls except for communication errors and RPC server or client failures. OSI RPC supports "at-most-once" call semantics in the first release of the standard where the RPC service provider ensures that the remote procedure is not executed more than once. The RPC service provider makes reasonable attempts to overcome failures but may eventually report a failure. In the case of a successful return, the caller knows that the remote procedure has been executed exactly once; however, in the case of a failure return, the client can not determine whether or not the remote procedure was executed.

OSI RPC supports callbacks. During the execution of the called remote procedure in the server, procedures in the same execution context of the calling procedure in the client (as defined in the interface definition) can be called back by the server using the same binding. The callback procedures may be regarded as child operations under parent operations.

OSI RPC includes the concept of RPC termination, which is the response generated by a particular remote procedure call request. For each request made upon a particular remote procedure, the response will be exactly one of a number of possible terminations defined for that procedure. Three classes of terminations are defined. The most common termination that is always possible is the return of the procedure's output and input/output parameters as defined in the application's interface definition. This is referred to here as the "normal" termination. The second class of terminations are called RPC-specific terminations which are defined in the OSI RPC standard. These address problems such as binding errors, marshalling errors, invalid context handles, insufficient resources, RPC cancelled, and invalid binding

handles. The third class is for system-specific terminations defined elsewhere, for example in other standards documents or by system implementors.

The concept of context handles resulted from the need for identifying state or context information across successive calls between different address spaces. A context handle allows a server availing multiple clients to distinguish state and resource usage that it maintains on behalf of a particular client. A context handle does not imply a recovery mechanism in the event the client or the server fails.

RPC cancel is an extension of local interrupts to the remote case. Locally, a process may receive an interrupt from another process, its environment or its user (for example, the user pressed Control/C to abort) requesting the process to terminate in an orderly fashion. To preserve the local semantics, if the process which receives the interrupt has outstanding RPCs, then the interrupt request should be propagated to each one of these RPCs requesting their orderly cancellation. The normal action taken by the RPC when receiving an interrupt is to terminate in an orderly fashion. Its response is either its output parameters or the termination "remote procedure call cancelled."

3.2 OSI RPC Requirements

The current draft includes a comprehensive set of RPC requirements. The list of requirements is too lengthy to introduce in this paper, however, we'll highlight some of the key requirements below.

- **General Requirements**

- Support the local procedural call construct of ISO programming languages; allow the caller and called programs to be written in different languages.
- Provide a high-level notation for describing the attributes specific to the RPC client or server.
- Allow multi-threading.

- **Requirements for the Interface Definition Notation (IDN) and Datatypes**

- Support parameter attributes of input, output or input/output.
- Allow future mapping to various ISO programming language binding.
- Support built-in datatypes: integer, real, boolean, enumerated, char, char string, numeric string, and procedure reference.
- Support full typed pointer.
- Support aggregate datatypes:
  - record with fields to be ignored by marshalling
  - discriminated union
  - conformant array with optional lower/upper bounds
  - context handle
- Support multibyte international character sets.

- **Service and Protocol Requirements**

- Support resilience from failure.
- Support omitted arguments.
- Support the signalling of an outstanding call.
- Support various binding policies at user option, e.g., binding before the time of the first call, binding at the time of the first call, rebind for each call.
- Allow future incorporation of ISO security and transaction processing.

In the following sections, we'll demonstrate how the OSF RPC implementations can satisfy these requirements.

3.3 OSI RPC Service Primitives

A set of RPC service primitives are specified in the current OSI RPC draft. They are abstract representations of interactions between the RPC service user (i.e., the stubs generated by the IDN compiler) and the RPC service provider. Note that these primitives do not represent application program interface routines as used by the programmers.

- RPC-BIND establishes an RPC binding between the invoking RPC client and the named RPC server using a named Interface. This involves locating the server (e.g., importing from a name server) and allocating a binding handle. This service can be invoked by the client only. The time that RPC-BIND service is invoked is determined by a user-specified binding policy.

- RPC-UNBIND performs an orderly release for the named RPC Binding. This service can be invoked by the client only.

- RPC-INVOKE is used to call a remote procedure using the specified RPC Binding, which may be supported by both synchronous and asynchronous programming interfaces.

- RPC-RESULT is used to return the input/output and output parameters at the end of an execution of the invoked remote procedure.

- RPC-ERROR is used to indicate an environmental or system-related error which occurred during the course of the RPC call. No input/output or output parameters are returned.

- RPC-CANCEL is used to interrupt an outstanding remote procedure call.

- RPC-REJECT is issued by the RPC service provider to indicate that an environmental error has been detected by the service provider.

3.4 OSI RPC Protocols

As defined in the RPC requirements, OSI RPC protocols will be delivered using OSI Remote Operations. The current draft specifies the rules for marshalling the parameters and function-result to the corresponding Argument field of the ROIV (Remote Operation Invoke) and RORS (Remote Operation Result) APDUs (Application Protocol Data Unit).

Detailed mappings between the RPC service primitives and the resulting ROSE protocols are not fully specified in the draft. We'll use the mappings defined in the ECMA RPC to illustrate the protocol exchanges involved in a typical scenario in which an RPC is invoked by a client and a normal procedure return is received from a server:

1) the client issues an RPC-INVOKE request, which is mapped onto an RO-INVOKE request and produces an ROIV APDU to be delivered to the server,
2) the server receives the ROIV APDU via an RO-INVOKE indication, which is mapped onto an RPC-INVOKE indication,
3) after the remote procedure completes, the server issues an RPC-RESULT request, which is mapped onto an RO-RESULT request and produces an RORS APDU to be delivered to the client,
4) the client receives the RORS APDU via an RO-RESULT indication, which is mapped onto an RPC-RESULT indication.

The Argument field of the ROIV APDU generally contains input parameters, input/output parameters, and output parameters or procedure function-result containing an array or string whose bounds are not all constants.

The Result field of the RORS APDU contains RPC status information and input/output parameters, output parameters, and function-result.

3.5 Interface Definition Notation

A working version of IDN grammar syntax consistent with the OSI RPC requirements is now included in the second working draft. The IDN consists of about eighty productions in extended BNF format. A sample of the first level breakdown of the IDN grammar syntax is provided below for demonstration purpose:

<interface> ::= interface <interface_identification>
begin <interface_body> end

<interface_body> ::= [<import>] <declaration>; [<declaration>;] ...

<import> ::= import (<import_list>)

<import_list> ::= <interface_identification> [, <interface_identification>] ...

<declaration> ::= <type_dcl> | <const_dcl> | <proc_dcl> | <exception_dcl>

An interface consists of a header and a body.

<type_dcl> associates an identifier with a newly specified type which may be constructed from simple datatypes or constructed datatypes. It also optionally specifies attributes (e.g., reference-only pointer) for the type being defined.

<const_dcl> associates an identifier with a value for certain restricted types (integer, real, char, and boolean). These values can be used in defining numeric ranges, discriminated unions, array bounds, and in the sparse attribute.

<proc_dcl> associates an identifier with some (optional) attributes (e.g., idempotent, callback), an (optional) return value, procedure parameters with directional attributes (e.g., in, out, inout), and an (optional) list of exceptions that the procedure can return. The procedure by default resides on the server which exports the interface, unless it is specified to be a client procedure using the callbacks or the residence attribute. In this case, it is a callback routine that resides on the client.

<exception_dcl> specifies the name of the exception and the parameters returned when this exception is raised. Note that the current model allows for exceptions to return parameters which may not be the same as those returned upon normal termination.

4 . Architecture of OSF RPC

4.1 Interface Definition

4.1.1 Object-oriented Approach

In an object-oriented system, all resources in a network are characterized as objects. Objects are organized into object types and are manipulated by well-defined operations (procedures). An interface is a set of related operations. Each object type has one or more interfaces associated with it. All accesses to a particular object are made through the operations defined in the interfaces associated with that object's type. The OSF RPC system enables distributed applications to be defined and built in an object-oriented way. Remote procedures are viewed as operations on objects (printers, files, computing engines, etc.) rather than as calls to particular machines or server processes. This approach to problem solving enables application designers to create programs that are less dependent on hardware configurations such as host names and locations.

4.1.2 Network Interface Definition Language

The OSF RPC utilizes the Network Interface Definition Language (NIDL) for the specification of RPC interfaces. NIDL consists of a subset of ANSI C syntax and semantics with additional constructs to address the requirements of the RPC interface. An interface definition describes the constants, types, operations and parameters for an interface. In short, an interface definition specifies the interface between a client of a service and the provider of the service; it defines how a client "sees" a remote service and how a server "sees" requests for the service it provides. Note that NIDL is strictly a declarative language; it contains no algorithmic, executable constructs (e.g., if-then-else, loop, compute).

The NIDL Compiler takes an interface definition written in NIDL as input and generates source code in portable C for client and server stub modules which are then integrated with the client and server applications respectively. The stubs produced by the NIDL Compiler handle and hide all the "remoteness" in a distributed application. They perform data encoding/decoding and marshalling/unmarshalling, and interacts with the RPC run-time system.

The NIDL grammar consists of a rich set of production rules. It would be too tedious to cover them in detail here. Instead we'll briefly summarize the key features using the OSI RPC requirements as a guideline.

- Support the local procedural call construct of ISO programming languages. NIDL allows mapping to various ISO programming language binding via the NIDL compiler directive.

- ACF (Attribute Configuration facility) allows a RPC user to specify attributes specific to the RPC client or server which are separate from the network contract. Examples are binding policies, code-generation options, etc.

- Support all parameter attributes and built-in datatypes required. In addition, NIDL allows constant declarations, customized type declarations (like 'typedef' in C), and predefined types such as error status. Type declarations in an interface can be exported to and imported from other interfaces.

- NIDL supports full typed pointer and allows full transparency over remote calls:
  - support NULL pointer value,
  - support change of a pointer value across a remote call (e.g., reflect node creation/deletion/orphaning and topology change made by remote procedures),
  - support aliasing

- Support all data aggregate defined, especially
  - record with fields to be ignored by marshalling. NIDL also allows parameters to be ignored as well.
  - discriminated union with the discriminant variable explicitly specified.
  - conformant array with optional lower/upper bounds. In addition, NIDL supports varying array, which allows only a slice of an array to be transmitted for efficiency purposes. NIDL also allows dynamic allocation size of an array using alloc attributes.
  - context handle is supported by a pointer to an opaque type with a context-handle attribute. It provides a mapping between a client and a server's address spaces. The server may provide a context handle-specific run-down routine if the run time detects a client failure.

- Support multibyte international character sets.

NIDL also supports many other useful value-added features beyond what are required by OSI RPC. One notable example is the "pipe" datatype. Pipe enables the transfer of very large quantities of typed data in an stream using two specialized callbacks (pull and push). Pipe has been proven to be very useful in transferring large chunks of data between files and in sending unrestricted-size of data between an open-ended producer and a receiver.

4.2 Run-time System

The OSF RPC run-time system (RTS) is divided into three major components as shown in Figure 3: the Communication Service, Naming Service, and Management Service. A brief description of each of these services follows.

- Communication service (CS). The CS is responsible for the actual work being done when a remote procedure is called. It accesses the network and provides the means for simultaneous execution of different transport and RPC protocols.

- Naming service (NS). The NS accesses the distributed system's naming service such as an X.500 directory to locate servers by specifying an interface or an object. To this end, these have to be exported by servers and imported by clients.

- Management service (MS). The MS manages RPC services either locally or remotely. Remote management functions are a subset of local ones and are made available via the RPC mechanism.

Each one of these components will be scrutinized in more detail in the following sections.

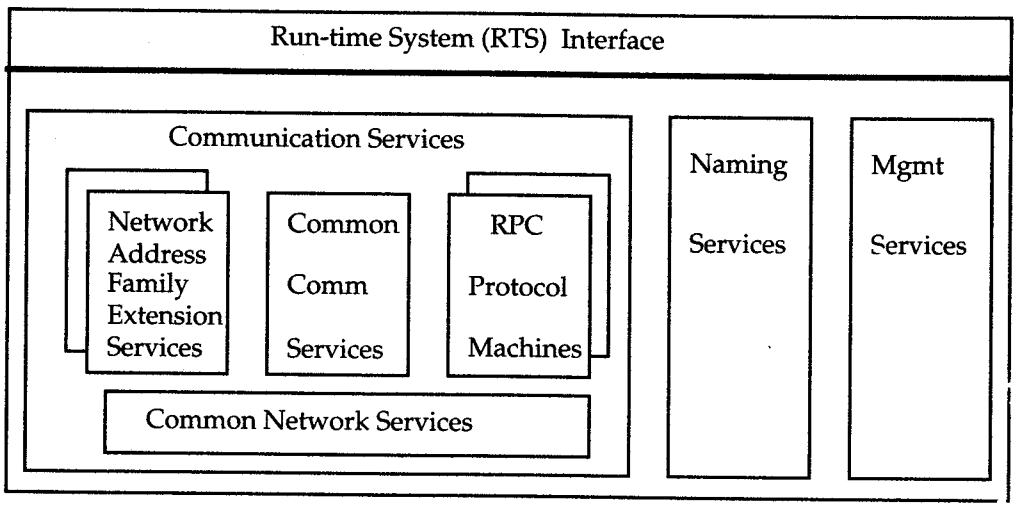

Figure 3 The OSF RPC Run-time System Architecture

4.2.1 Communication Service

The communication service (CS) allows for the use of different network and transport services as well as different RPC protocols.

The CS contains components that are common to all combinations of transport and RPC mechanisms. Some additional ones are specific to each protocol. To the former belong the common communication service (CCS) which will be explored later and the common network service which contains network-independent functions such as read, write, open, close. There are, however, network- dependent features - for example, functions for manipulating addresses. These dependencies are covered up in separate modules, one for each network service. These so-called network address family extension services all present the same interface to the CCS. For that reason, whenever a network-specific operation such as returning the endpoint in a network address needs to be performed, CCS makes the corresponding function call. This call results in the activation of one network-specific module; which one is determined during initialization time.

The hiding of the two supported RPC protocols, connection or datagram based, is achieved in the same way. For each RPC protocol machine, an initialization routine returns function pointer arrays which present a common interface to the consuming CCS. The latter uses them to make the data transfer calls, collect statistics, access the relevant fields in the binding representation structure, and notify the RPC protocol machine of network events.

This flexible architecture allows the inclusion of further transport or RPC protocols in a well-structured manner. For example, a ROS-based RPC protocol could fit here.

4.2.1.1 Common Communication Service (CCS)

A brief overview of the services constituting the CCS shown in Figure 4 is provided in the following sections.

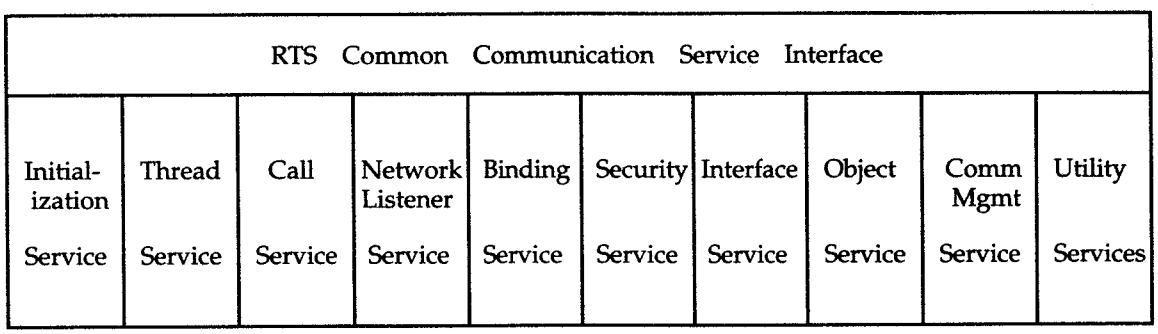

Figure 4 The Common Communication Services

4.2.1.1.1 Initialization Service

The RPC environment is set up in the initialization service; prerequisites for common communication services such as the allocation of static tables are created there as well. Transport and RPC protocols to be supported are also determined during initialization. A table is generated that assigns an identifier to each combination of network, transport, and RPC protocols (RPC protocol sequence id).

Identifiers of this kind appear later as attributes to structures that specify interfaces; a so-called interface representation data structure may exist several times for the same interface but with different protocol sequence identifiers associated with it. It follows that the same interface can be reached via different protocol stacks.

If the RTS library is built as a non-shared image, the initialization routines are called when either the application or the stub first calls the CS.

4.2.1.1.2 Thread Service

As in the case of local procedure call, a single thread of execution exists logically when a remote procedure call is performed. A client makes its request, the stub and RTS code on both sides is executed, the actual operation gets done, and the result is transferred back to the client.

In the OSF RPC implementation there is a client call thread for each request an application makes. This thread executes all of the stub and CS code. When all the parameters are marshalled and passed to the network, the client thread is blocked until the result of the call is returned.

Inside the CCS runs the network listener thread which has the same life time as the entire RPC process. It detects events on network descriptors, such as socket descriptors; these events are forwarded to a dispatcher routine which assigns them to an RPC PM and unblocks the

associated client thread. The latter continues fetching all the data pertaining to the result of the original request and returns to the calling application.

On the server's side a listener thread waits for events on the network. Upon reception of a new operation, the dispatcher determines which RPC protocol to use and starts a server call thread. This thread accepts the data, does the unmarshalling , performs the actual operation and returns the result to the network. When all the data are marshalled and sent the server call thread terminates itself.

The thread service component of the CCS contains functions for manipulating threads such as creating and destroying them; furthermore, a table is managed that keeps track of a thread's status and relates it to the RPC context it is executing.

4.2.1.1.3 Call Service

The call service provides functions send and receive data. Furthermore, it manipulates the call handle data structure which serves as a parameter for all the functions in this service. The important components of this structure are the binding representation structure which contains information about the remote partner, interface and operation identification, a thread identifier, dynamic information such as the employed transfer syntax, and RPC protocol-specific information. The other parameter common to the call service functions is the so-called I/O vector, which is an array of buffer descriptors. The buffers pointed to in the I/O vector provide the memory for the marshalled data.

The routines in the call service are used to start and end transfers or to send and receive data. Data is sent or received in fragments; for larger quantities of data these calls have to be repeated until all the data is processed. There is a special call that is utilized to indicate to the remote RPC process that the last chunk of data belonging to the same request or response is being sent.

For exceptional situations, there are calls to send fault messages to the remote RPC or to alert the remote thread. The latter may result in the cancellation of the procedure being performed by a server.

4.2.1.1.4 Network Listener Service

The network listener service detects events on any network and delivers them to the correct RPC PM. To this end it accesses an internal table where network descriptors are related to RPC protocol sequences (transport and RPC protocols). It manages that table (add, delete operations) as well as monitors the network itself. For example it checks the liveness of the remote partner in the case a connectionless protocol is employed. The network listener service further manages the network listener thread (creation, termination, notification of events).

4.2.1.1.5 Binding Service

The CCS binding service provides functions to manipulate the binding representation data structure or binding handle. A client makes a request to the naming service to first obtain this structure (see the RPC RTS Naming Service in Section 4.2.2). The binding handle contains information that relates a client to the server process. It consists of the location of the server (the entire presentation address consisting of the server network address and the communication port of the server process), an object identifier, an interface identifier and further RPC protocol specific information. This structure is pointed to in the call handle which is part of every function in the call service.

4.2.1.1.6 Security Services

Besides allowing no security at all, the OSF RPC allows for four levels of security to be selected: 1) perform authentication on every association establishment, 2) perform authentication on every call, 3) enforce integrity on every packet, and 4) enforce privacy on

every packet. Encrypting and decrypting user data and checksums in RPC message packets will provide data privacy (confidentiality) and data integrity.

Currently the Kerberos authentication scheme using secret-key is employed. However, the flexible design makes the underlying security mechanism transparent to the OSF RPC users. For example, a public-key based security mechanism can be used transparently in the future without affecting the existing RPC-based applications using Kerberos.

4.2.1.1.7 Interface Service

The functions in the interface service are used to handle the internal interface registry table which keeps track of all the interfaces registered with the CCS. It contains interface uuids, version numbers and operation counts.

4.2.1.1.8 Object Service

The object service manipulates another internal table that relates objects to the types to which they belong. Particular objects are servers, server groups and configuration profiles.

4.2.1.1.9 Communications Management Service

The functions in this service allow to manage all of the CS: CCS, network services, and RPC protocol services. This includes functionalities such as stop processing incoming calls, indicate how fault messages should be treated, allow or bar remote shut-downs, and gather statistics.

4.2.1.1.10 Utility Service

This component subsumes common services for handling uuid, timers, and buffers.

4.2.1.2 Connection-oriented RPC protocol service

As mentioned before, the OSI RPC standard will utilize OSI ACSE, ROSE, and presentation service. The design of OSF's connection-oriented RPC is consistent with the OSI RPC direction. This implies that a presentation context is negotiated when a association between a client and a server is established; PDUs exchanged resemble those in the OSI standards. Each interface supported by a server, in OSI terminology, corresponds to a different abstract syntax. Version numbers are provided so that a new interface version does not require a new abstract syntax to be defined. Currently the only transfer syntax supported by the OSF RPC is NDR. This means that the RPC control information in a message header is encoded in NDR without utilizing padding or fields for type and length. The body part of a message that represents marshalled data can be encoded in any transfer syntax as desired.

The OSF RPC run-time system maintains associations in pools such that associations can be reused for remote procedure calls between the same client-server pair. Thus allocating an association for processing a remote procedure call either leads to establishing a new association or to reusing a pre-existing association when one is available.

For servers supporting more than one interface, it is possible to negotiate a set of presentation contexts during association establishment time. Reusing an existing association for a new interface necessitates the modification of the presentation context which is achieved by utilizing P-ALTER-CONTEXT functions in the presentation layer. Policies for the number and lifetime of associations are defined to reflect the usage of RPC. At least one association has to be maintained between a client-server pair as long as there is an active context handle between the pair.

The connection-oriented RPC protocol allows for fragmentation of request, response and error messages; maximum sizes must be negotiated as part of the RPC protocol during connection establishment.

4.2.2 Naming Service

The naming service provides a naming service independent interface allowing both a server to export its services and a client to import services it desires. This interface allows for accessing and managing data stored in a global directory provided it supports a hierarchical name space and multivalued attributes. The OSF RPC uses logical names to access entries in the name space. The entries are classified into server, server group, and profile entries. Mandatory attributes of a server entry include a unique interface identifier, an interface version, a data representation, an RPC protocol version, and a binary handle for the server. Optionally, there may be further unique identifiers that reference objects such as files supported by the server. It is possible that a server offers more than one interface. This requires that the above-mentioned attributes occur more than once; thus arises the need for multiple attribute values in the name service.

A server group represents a collection of servers, whereas a profile is a reference to servers or server groups. By specifying a profile for import or look-up operations, a path through the name space database which is used to search for a particular server entry is determined. The starting entry from which to begin the search must be specified separately; this can be done explicitly or by interpreting an environment variable.

The RPC naming service provides the following groups of services:

- operations for manipulating binding handles (e.g., export, import, lookup). None of these operations imply communications between the client and the server.
- services for retrieving information about entries in general (e.g., interfaces and objects exported by a server).
- operations on server groups (e.g., create, delete, add, remove members).
- operations on profiles.
- interfaces for management applications for creating or deleting name service entries in a consistent way.

4.2.3 Management Service

The management services are grouped into two classes: 1) functions that can only be used by an application to manage itself locally, and 2) functions that can be called to manage remote RPC-based applications. For successful remote management, an application to be managed has to make itself available by calling appropriate run-time functions. An application can control remote access to its management interface through authenticated RPC services.

Examples for local management services are calls to inquire about or set time-out values for binding handles or alerts. Remote management functions are used to determine whether a server is listening for incoming calls, start/stop listening on a server, or collect statistics.

5. RPC as Used in the OSF Distributed Computing Environment

Using RPC as a key building block, OSF has integrated a set of enabling distributed services into an architecturally coherent system that provides a layer of logical simplicity to mask the physical complexity of the distributed environment. The goal of the OSF's Distributed Computing Environment (DCE) is to provide a comprehensive offering of core distributed services that support the development, use and maintenance of distributed applications in a heterogeneous networked environment.

The OSF DCE includes the following services that can be utilized, separately or in combination, to build distributed environments and applications by end users or application developers.

- RPC, including presentation service. (as described in this paper)

- Naming (Directory) service. To allow both users and applications to store, retrieve, and manage information about objects by human sensible, hierarchical names within a global context. Both a local name server and a global X.500 directory are provided.

- Security service. To prevent unauthorized access to resources by providing user registration and authentication, data integrity and confidentiality, and access control list management.

- Threads service. To provide a means of creating and managing multiple threads of execution within a single user process, allowing programmers to build parallelized client and server applications in a multi-party distributed environment.

- Time service. To synchronize the clocks in all the component systems of a distributed environment so that applications can ascertain the order of network events, compute the interval between events, and schedule events independently of locations.

- Distributed file system. To provide the ability to store and access data at remote locations by extending the local file system model to the network, allowing users full access to remote files within a global context. This distributed file system is characterized by high availability, high scalability to very large configurations, and secure access to data.

- PC integration service. To extend the PC MS-DOS environment by providing access to file services and print services that are exported by non MS-DOS server systems.

In addition to providing services for end user applications, these component services also provide essential services for each other and thus enables them to integrate together as a whole within DCE. Figure 5 describes the interrelationship between RPC and other components within the OSF's DCE. For example, the RPC run-time system has been designed to take advantage of other DCE component services in the following way:

- Threads service - for concurrent processing of multiple incoming and outgoing calls.
- Time service - for constructing universally unique identifiers and synchronizing events.
- Naming service - for exporting and importing RPC interfaces, and
- Security service - for RPC client-server mutual authentications, data integrity and privacy.

Likewise, RPC is used internally as the only means of communications between the client and the server in other DCE component services, namely:

- Time service,
- Distributed file system,
- Naming service , and
- Security service

The OSF RPC has been proven in meeting diverse needs in these DCE services which are all based on the client-server model. Examples are fast response time required by time-critical services such as name lookup and time synchronization, high throughput required by moving large amounts of data in distributed file replication and backup, concurrent and secure calls for highly-demanded file and name servers, just to name a few. The benefits of using RPC as discussed in this paper have also been evidenced in our experience: the OSF RPC hides the network complexity and system heterogeneity from programmers and thus shortens our development cycle; it also facilitates distributed development and maintenance of client and server applications based on well-defined network contract specified in NIDL. Moreover, by integrating RPC with other DCE services, we have demonstrated that RPC is the key building block in the infrastructure required in the development of today's distributed environments and applications.

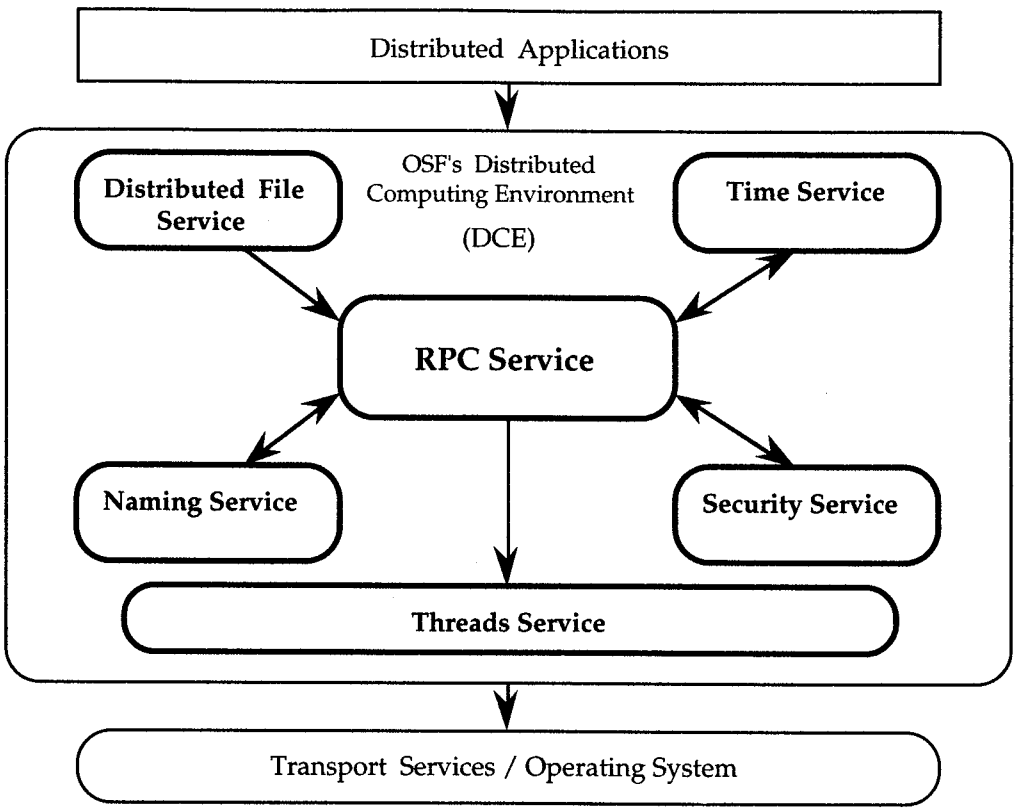

Figure 5 OSF's Distributed Computing Environment

Note. The arrow line in the diagram indicates the service user-provider relationship.

6. Conclusion

We have demonstrated in this paper that a state-of-the-art RPC system fulfills the expectations and meets the demand for the development of today's distributed applications. We envision that the RPC-based programming practice will be growing rapidly in importance and popularity in the 90's. The OSF RPC presents a feasible engineering realization for the computational model and requirements specified in the current OSI RPC draft. It is technically capable of allowing a smooth transition to the future OSI RPC standard as it is fully developed.

Because the OSF RPC NIDL is a functional superset of the semantics currently required for the OSI RPC IDN, those interfaces defined using either the OSI RPC IDN syntax or the ISO-equivalent subset of the OSF RPC NIDL syntax should be easily interchangeable. Advanced features in the OSF RPC will be treated as value-added features over the emerging OSI RPC standard.

The OSI RPC protocol stack will be supported as an additional RPC protocol machine under the well-defined OSF RPC Run-time interface. The ability to select different RPC protocols in the OSF RPC at run time will ensure a smooth migration path between current OSF RPC-based applications and future OSI RPC-based applications.

This paper was originally published in the proceedings of ONLINE '91, Hamburg, Germany.
©Open Software Foundation, Inc. 1991, 1992.