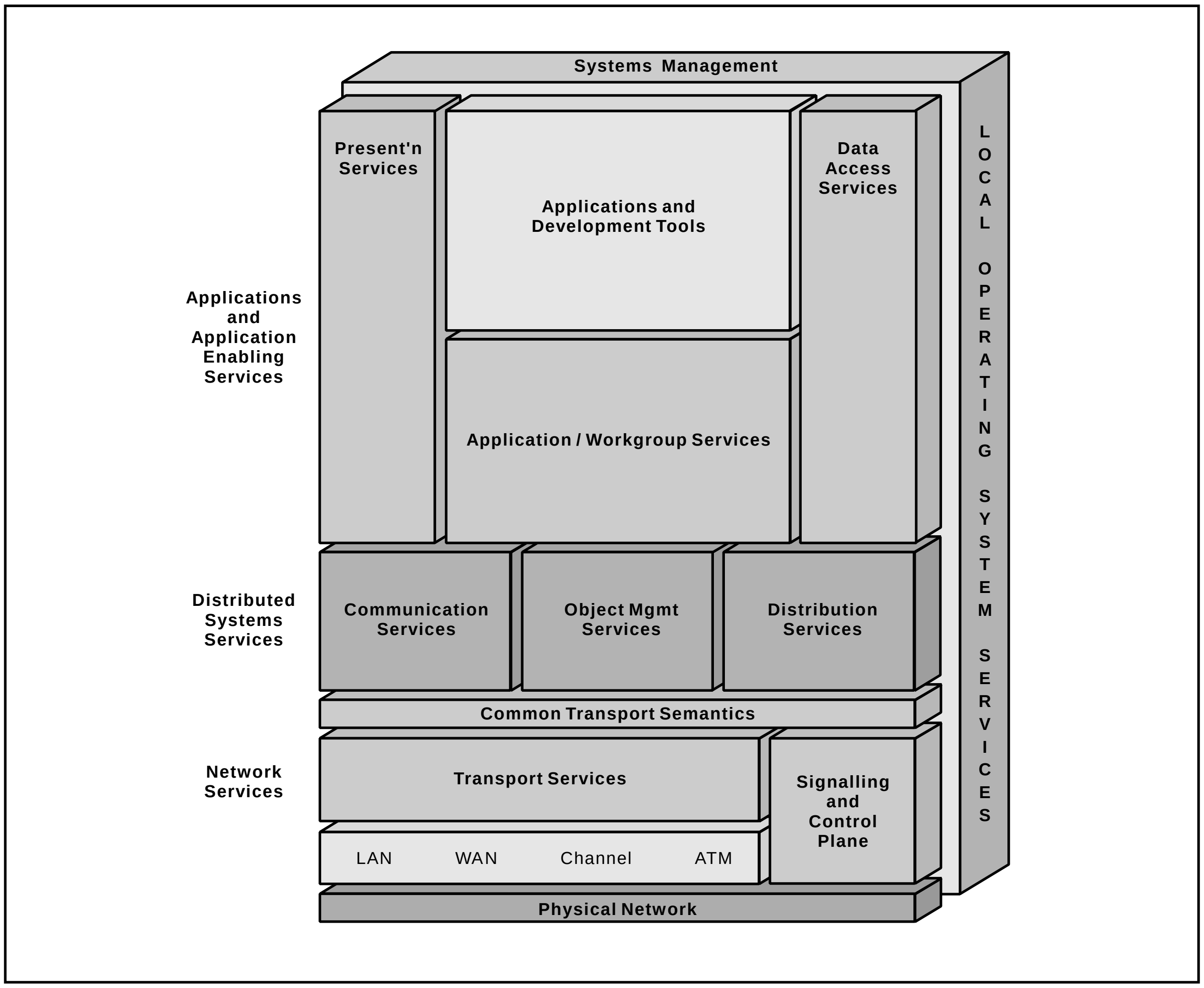
Open Blueprint

Remote Procedure Call Resource Manager



Systems Management
Present'n Services
Applications and Development Tools
Data Access Services
LOCAL OPERATING SYSTEM SERVICES
Applications and Application Enabling Services
Application / Workgroup Services
Distributed Systems Services
Communication Services
Object Mgmt Services
Distribution Services
Common Transport Semantics
Network Services
Transport Services
Signalling and Control Plane
LAN
WAN
Channel
ATM
Physical Network

GC23-3926-01

Open Blueprint

IBM

Remote Procedure Call Resource Manager

GC23-3926-01

About This Paper

Open, distributed computing of all forms, including client/server and network computing, is the model that is driving the rapid evolution of information technology today. The Open Blueprint structure is IBM's industry-leading architectural framework for distributed computing in a multivendor, heterogeneous environment. This paper describes the RPC resource manager component of the Open Blueprint and its relationships with other Open Blueprint components.

The Open Blueprint structure continues to accommodate advances in technology and incorporate emerging standards and protocols as information technology needs and capabilities evolve. For example, the structure now incorporates digital library, object-oriented and mobile technologies, and support for internet-enabled applications. Thus, this document is a snapshot at a particular point in time. The Open Blueprint structure will continue to evolve as new technologies emerge.

This paper is one in a series of papers available in the *Open Blueprint Technical Reference Library* collection, SBOF-8702 (hardcopy) or SK2T-2478 (CD-ROM). The intent of this technical library is to provide detailed information about each Open Blueprint component. The authors of these papers are the developers and designers directly responsible for the components, so you might observe differences in style, scope, and format between this paper and others.

Readers who are less familiar with a particular component can refer to the referenced materials to gain basic background knowledge not included in the papers. For a general technical overview of the Open Blueprint, see the *Open Blueprint Technical Overview*, GC23-3808.

Who Should Read This Paper

This paper is intended for audiences requiring technical detail about the RPC Resource Manager in the Open Blueprint. These include:

- Customers who are planning technology or architecture investments
- Software vendors who are developing products to interoperate with other products that support the Open Blueprint
- Consultants and service providers who offer integration services to customers

Contents

Figures

Introduction

A distributed application based on the client/server model consists of two parts: the client side of the application, which runs on one system and makes a request for service on behalf of a user, and the server side of the application, which runs on another system on the network and fulfills the service request. The two pieces of code on two different systems need to be able to communicate across the network. A model for implementing communications between the client and server of an application is the remote procedure call (RPC).

RPC gives programs the ability to express an interaction between the client and server of a distributed application as if it were a local procedure call: the program defines a calling interface and a procedure that implements it, makes a call to the procedure along with any arguments, and receives a return value through the argument list or as the procedure result.

In RPC, as in a traditional local procedure call, control is passed from one code segment to another, and then returns to the original segment. However, in a local procedure call, the code segments are in the same address space on the same system; whereas in a remote procedure call, the called procedure runs in a different address space, usually on a different system than the calling procedure. As a result, arguments and results are passed differently for local and remote procedure calls. In local procedure calls, arguments and return values can be passed on the process's stack. In remote procedure calls, arguments and return values must be packed up into messages and sent to the remote system over the network. The underlying RPC mechanism makes this look like a procedure call to the program.

An RPC facility shields the application program from the details of network communications between client and server systems, such as:

- Marshaling and unmarshaling of parameters
- Fragmentation and reassembly of messages
- Handling different data formats (such as byte ordering) between different systems
- Using a directory service to find message recipients
- Using security services to ensure secure communications

Programs using RPC do not need to be rewritten in order to port them to different architectures, operating systems, communications protocols, or languages. RPC provides a high-level programming model for the distributed application program, hides communications details, and removes non-portable system and hardware dependencies.

An RPC application uses dispersed computing resources such as central processing units (CPUs), databases, devices, and services. The following are examples of RPC applications:

- A calendar-management application that allows authorized users to access the personal calendars of other users.
- A graphics application that processes data on central CPUs and displays the results on workstations.
- A manufacturing application that shares changing information about assembly components among design, inventory, scheduling, and accounting programs located on different computers.

The end user does not come in direct contact with RPC, but does see the end result, in the form of:

- The availability of distributed applications built using RPC
- The ability to use remote resources accessed through RPC

An end user who is browsing through the namespace may also notice the names of RPC-based servers, since these servers advertise themselves to their clients through the Open Blueprint Directory Service.

The Open Blueprint RPC resource manager is based on OSF DCE RPC. References to “directory” in the remainder of this paper refer to Open Blueprint Directory Service.

What Is RPC?

RPC is a facility for calling a procedure on a remote system as if it were a local procedure call. A remote call appears similar to a local call, but there are several RPC components that work together to implement this facility. These facilities include the Interface Definition Language (IDL) and its compiler, a Universal Unique Identifier (UUID) generator, and the RPC Runtime. The RPC Runtime uses the Open Blueprint Common Transport Semantics. RPC supports connectionless and connection-oriented transports.

The components that comprise the RPC are:

- The Interface Definition Language (IDL) and its Compiler

  An RPC interface is described in IDL. The IDL file is compiled into object code using the IDL compiler. The object code is in two main parts - one for the client side of the application, and one for the server side.

- The Transfer Syntax

  This protocol defines rules for encoding data in such a way that the communicating systems, which may or may not have differing internal data representation formats, can understand each other. The Transfer Syntax is handled by the RPC Runtime library and IDL-compiler-generated stub code.

- The RPC Runtime Library

  This library consists of a set of routines, linked with both the client and server sides of an application, which implement the communications between them. This involves the client finding the server in the distributed system, getting messages back and forth, managing any state that exists between requests, and processing any errors that occur.

- Authenticated RPC

  RPC is integrated with the Open Blueprint Identification & Authentication resource manager (or component) to provide secure communications. Levels of security can be controlled by the RPC application program through the Authenticated RPC API.

- Name Service Independent (NSI) API

  RPC is integrated with the Open Blueprint Directory Service component to facilitate the location of RPC-based servers by their clients. The NSI routines allow a program to control the association, or binding, of a client to a server during RPC.

- The Host Daemon

  The Host daemon is a program that runs on every system. It includes (among other things) an RPC-specific name server called the endpoint mapper service, which manages a database that maps RPC servers to the transport endpoints (in IP, the ports) on which the server is listening for requests.

- The Control Program

  The control program is a tool for administering RPC. It also allows an administrator to access RPC data in CDS.

- UUID Facilities

  These are ancillary commands and routines for generating Universal Unique Identifiers (UUIDs), which uniquely identify an RPC interface or any other resource.

What is Transactional RPC?

Transactional RPC allows programs to use RPC for transactions in a distributed environment. It works in conjunction with the Open Blueprint Transaction Manager resource manager (synchronizer) to synchronize the interaction of transactions that may be submitted from multiple locations on a network and allow transactions to be distributed to other programs and systems while maintaining transaction semantics such as two-phase commit. Transactional RPCs carry additional information about the transaction, for example, the ID of the transaction on whose behalf they are acting. Transactional RPCs guarantee that the remote procedure to which the call is placed is executed exactly once if the transaction commits and not at all if the transaction aborts. Its components include:

- The Transaction Interface Definition Language (TIDL) and its Compiler

  An TRPC interface is described in TIDL which provides transactional enhancements to RPC IDL. TIDL allows programs to more easily express the additional information required by the transactional RPC. The TIDL file is compiled into an IDL file and object code using the TIDL compiler. The object code is in two main parts - one for the client side of the application, and one for the server side.

- The Transactional RPC Runtime Library

  TRPC adds a set of routines to the RPC Runtime Library, linked with both the client and server sides of an application, which provide procedures such as appending or retrieving the additional information needed for transactions. This transactional information is inserted into and removed from RPC requests as the request is passed from client to server (and likewise on the response). TRPC uses the RPC runtime for communicating the request/response between systems.

Remote Procedure Call Model

The Remote Procedure Call (RPC) model is a well-tested, industry-wide framework for distributing applications.

- Procedure Calls

  The model is derived from the programming model of local procedure calls and takes advantage of the fact that every procedure contains a procedure declaration. The procedure declaration defines the interface between the calling code and the called procedure. The procedure declaration defines the call syntax and parameters of the procedure. All calls to a procedure must conform to the procedure declaration.

  A RPC executes a procedure located in a separate address space from the calling code. Applications that use remote procedure calls look and behave much like local applications. However, an RPC application is divided into two parts: an RPC server, which offers one or more sets of remote procedures, and an RPC client, which makes remote procedure calls to RPC servers. A server and its clients generally reside on separate systems and communicate over a network. RPC applications depend on the RPC runtime which controls network communications for RPC applications. The RPC runtime supports additional tasks, such as finding servers for clients and managing servers.

- Language

  RPC is language independent in the sense that the stubs generated by the IDL compiler can be called by programs written in any traditional programming language, provided that the language follows the calling conventions that the stub expects. The IDL compiler generates stubs that use the C language calling conventions. A client written in FORTRAN, for example, can call an IDL stub in the same way that it calls any local C procedure. It can then make a remote call to a server (possibly written in another language) that contains the server stub generated from the same IDL file as the client stub.

- Data Representation

  A transfer syntax enables communications between systems that represent local data differently. The default data representation format is the DCE Transfer Syntax, which is currently the Network Data Representation (NDR). NDR encodes data into a byte stream for transmission over a network. It allows various representations for different types of data, including multiple encodings for characters, integers, and floating-point numbers. It is multicanonical; that is, there are several canonical formats that can be used. The sender chooses one of these formats (in most cases, it will be the sender's native data representation), with information about what representation it has chosen. The receiver transforms the data into its own format if it is different from the sender's format. This model optimizes for the case when both sender and receiver use the same data representation, a frequent occurrence.

  The RPC communications protocols support the negotiation of transfer syntax to allow the support of different transfer syntaxes in clients and servers.

  The RPC architecture allows the use of transfer syntaxes other than DCE Transfer Syntax. For example, data could be formatted according to the ISO ASN.1/BER specification and sent in that format. Because RPC uses the DCE Transfer Syntax, distributed applications are system independent. DCE Transfer Syntax allows systems to transfer data even when their native data representations are not the same.

- Protocol

  RPC is independent of transport protocols; instead, it includes two different RPC protocols. The first runs over connection-oriented transport protocols; the second runs over connectionless (datagram) transport protocols. The program can specify the underlying RPC protocol, but the semantics of RPC calls are the same whether RPC is running over a connectionless or connection-oriented transport. Another RPC protocol could be used in place of these two RPC protocols.

RPC Objects

RPC enables clients to find servers that offer specific RPC objects. An RPC object is an entity that an RPC server defines and identifies to its clients. Frequently, an RPC object is a distinct computing resource such as a particular database, directory, device, process, or processor. Identifying a resource as an RPC object enables an application to ensure that clients can use an RPC interface to operate on that resource. An RPC object can also be an abstraction that is meaningful to an application such as a service or the location of a server.

RPC objects are defined by application code. The RPC runtime provides substantial flexibility to applications about whether, when, and how they use RPC objects. RPC applications generally use RPC objects to enable clients to find and access a specific server. When servers are completely interchangeable, using RPC objects may be unnecessary. However, when clients need to distinguish between two servers that offer the same RPC interface, RPC objects are essential. If the servers offer distinct computing resources, each server can identify itself by treating its resources as RPC objects. Alternatively, each server can establish itself as an RPC object that is distinct from other instances of the same server.

RPC objects also enable a single server to distinguish among alternative implementations of an RPC interface, as long as each implementation operates on a distinct type of object. To offer multiple implementations of an RPC interface, a server must identify RPC objects, classify them into types, and associate each type with a specific implementation.

The set of remote procedures that implements an RPC interface for a given type of object is known as a *manager*. The tasks performed by a manager depend on the type of object on which the manager operates. For example, a manager of a queue-management interface may operate on print queues, while another manager may operate on batch queues.

Universal Unique Identifiers

A Universal Unique Identifier (UUID) is a hexadecimal number. Each UUID contains information, including a timestamp and a host identifier. Applications use UUIDs to identify many kinds of entities. RPC identifies several uses of UUIDs, according to the kind of entities each identifies:

- Interface UUID

  Identifies a specific RPC interface. An interface UUID is declared in an RPC interface definition and is a required element of the interface.

- Object UUID

  Identifies an entity for an application; for example, a resource, a service, or a particular instance of a server. An application defines an RPC object by associating the object with its own UUID (known as an object UUID). The object UUID exists independently of the object, unlike an interface UUID. If different servers offer the same RPC object, the servers typically use different object UUIDs to identify it. A given object UUID is meaningful only while a server is offering the corresponding RPC object to clients.

To distinguish a specific use of an object UUID, a UUID is sometimes labeled for the entity it identifies. For example, an object UUID that is used to identify a particular instance of a server is known as an instance UUID.

A server usually generates UUIDs for its objects as part of initialization.

- Type UUID

  Identifies a set of RPC objects and an associated manager.

Servers can create both object and type UUIDs. UUIDs are generated using the RPC UUIDGEN utility.

The RPC Application

Figure 1 on page 8 summarizes the pieces that compose an RPC application and how those pieces fit together. This section discusses this composition.

RPC Application Code

An RPC server or client contains application code, one or more RPC stubs, and a copy of the RPC runtime. RPC application code is the code written for a specific RPC application. Application code implements and calls remote procedures, and also calls any RPC runtime routines the application needs. An RPC stub is an interface-specific code module that uses an RPC interface to pass and receive arguments. A server and a client contain complementary stubs for each RPC interface they share. The RPC runtime manages communications for RPC applications. In addition, the RPC runtime supports an Application Programming Interface (API) used by RPC application code to call runtime routines. These runtime routines enable RPC applications to set up their communications, manipulate server's information, and perform optional tasks such as remotely managing servers and accessing security information.

RPC application code differs for servers and clients. Minimally, server application code contains the remote procedures that implement one RPC interface; the corresponding client contains calls to those remote procedures.

The IDL File

The IDL file defines all aspects of an interface that affect data passed over the network between a client and a server. Client and server incorporate the same interface definition to communicate.

First, the program defines the RPC interface and associated data types using the Interface Definition Language (IDL). An interface is a group of operations that a server can perform. This grouping is similar to a module or library in a conventional programming language—a group of operations defined in a single file or unit. For example, a Bank Service might perform operations to debit, credit, or read the balance of an account. Each of those operations and their parameters must be defined in the IDL file. The collection of Bank Service operations (debit, credit, read balance) together form the Bank Service interface.

The process of defining RPC operations is similar to defining the input and output of a local procedure call, except in IDL only the calling interface is defined, not the implementation of the procedure (an IDL interface definition is similar to an ANSI C prototype definition).

Next, the IDL file is compiled using the IDL compiler. The compiler produces output in the C programming language and then calls the appropriate C compiler to produce object code. The output of the compilation consists of a client stub, a server stub, and a header file. The client and server stubs are routines that make the remoteness of the operation transparent to the client or server.

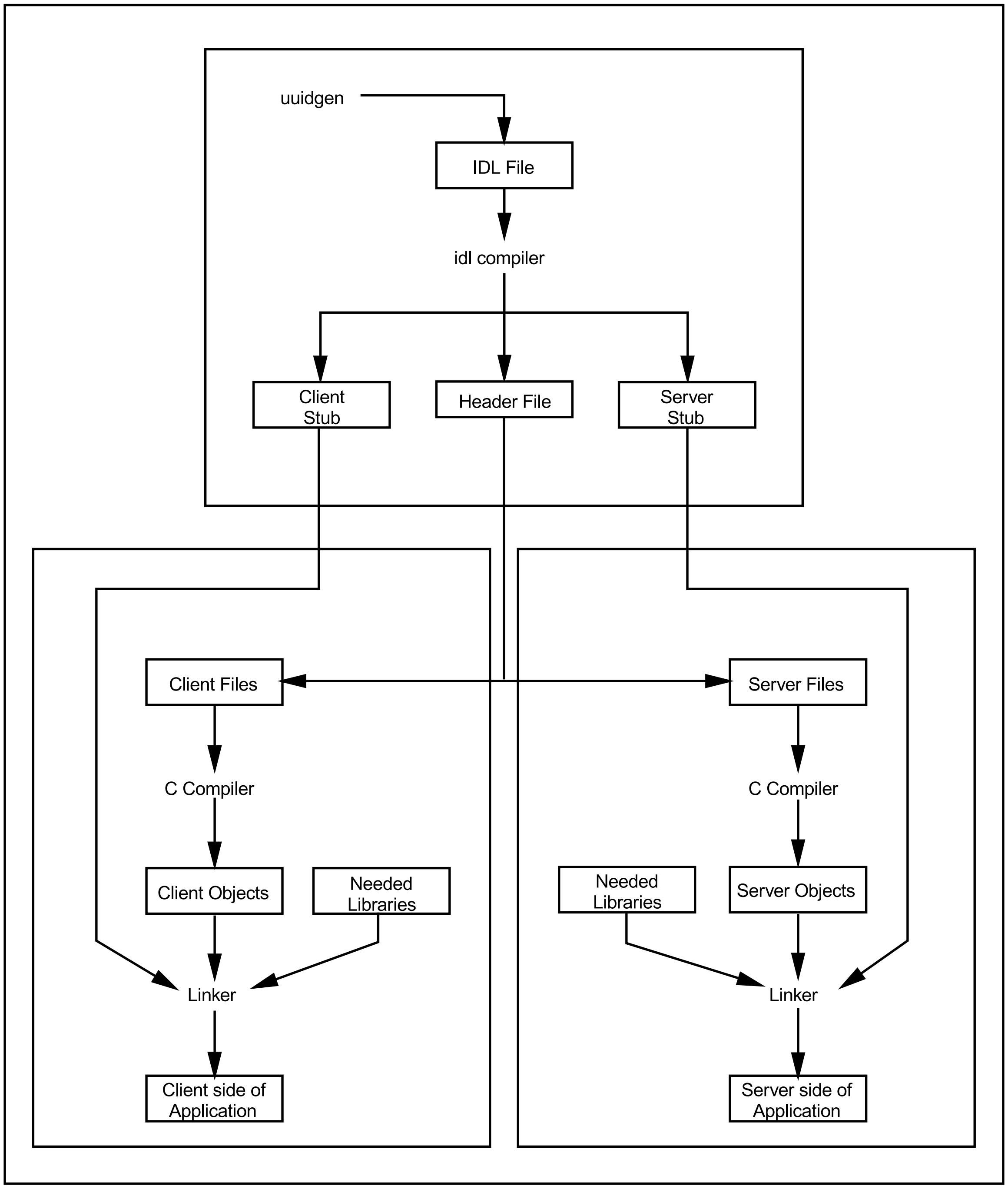


Figure 1. Composition of an RPC Application

Stubs

The stub performs basic support functions for remote procedure calls. For instance, stubs prepare input and output arguments for transmission between systems with different forms of data representation. The stubs use the RPC runtime to send and receive remote procedure calls. The client stub can also use the runtime to find servers for the client.

When a client application calls a remote procedure, the client stub first prepares the input arguments for transmission. This process is known as marshaling. Marshaling converts call arguments into a byte-stream format and packages them for transmission. Upon receiving call arguments, a stub unmarshals them. Unmarshaling disassembles incoming network data and converts it into application data using a format that the local system understands. Marshaling and unmarshaling both occur twice for each remote procedure call; that is, the client stub marshals input arguments and unmarshals output arguments, and the server stub unmarshals input arguments and marshals output arguments. Marshaling and unmarshaling permit client and server systems to use different data representations for equivalent data. The IDL compiler (a tool for distributed application development) generates stubs by compiling an RPC interface definition written by application developers. These stubs call the marshaling and unmarshaling routines for IDL data types in the RPC stub runtime.

To build the client for an RPC application, the client application code is linked to the client stubs of all the RPC interfaces the application uses. To build the server, the server application code is linked to the corresponding server stubs.

The RPC Runtime

In addition to one or more RPC stubs, every RPC server and RPC client is linked with a copy of the RPC runtime, that is, the RPC resource manager. Runtime operations perform tasks such as controlling communications between clients and servers and finding servers for clients on request. An application's client and server stubs exchange arguments through their local RPC runtimes. The client runtime transmits remote procedure calls to the server. The server runtime receives the calls and dispatches each call to the appropriate server stub. The server runtime passes the call results to the client runtime. The RPC runtime supports the RPC API (described in “Programming Support” on page 14) used by RPC application code to call runtime routines.

Server application code must also contain server initialization code that calls RPC runtime routines when the server is starting up and shutting down. Client application code can also call RPC runtime routines. Server and client application code can also contain calls to RPC stub-support routines. Stub-support routines allow applications to perform programming tasks such as allocating and freeing memory storage. Applications may have to manage parameter memory in a way that differs from the usual local procedural call semantics. This usually occurs in applications that pass pointer parameters that change value during the course of the call.

Execution Semantics

Execution semantics identify the ability of a procedure to execute more than once during a given remote procedure call. The communications environment that underlies remote procedure calls affects their reliability. A communications link can break for a variety of reasons such as a server termination, a remote system crash, a network failure. All invocations of remote procedures risk disruption due to communications failures. However, some procedures are more sensitive than others to such failures, and the impact of this sensitivity depends partly on how reinvoking the operation affects its results.

To maximize valid outcomes for its operations, the operation declarations of an RPC interface definition indicate the effect of multiple invocations on the outcome of the operations. Figure 2 summarizes the execution semantics for RPC calls.

Figure 2. Execution Semantics for RPC Calls

Semantics	Meaning
at-most-once	The operation must execute either once, partially, or not at all. For example, adding or deleting an appointment from a calendar can use **at-most-once** semantics. This is the default execution semantics for remote procedure calls.
idempotent	The operation can execute more than once; executing more than once using the same input arguments produces identical outcomes without undesirable side effects. For example, an operation that reads a block of an immutable file is *idempotent*. RPC supports **maybe** semantics and **broadcast** semantics as special forms of idempotent operations. maybe — The caller neither requires nor receives any response or fault indication for an operation, even though there is no guarantee that the operation completed. An operation with **maybe** semantics is implicitly **idempotent** and must lack output parameters. broadcast — The operation is always broadcast to all systems on the local network, rather than delivered to a specific system. An operation with **broadcast** semantics is implicitly **idempotent**.
exactly-once	The operation must execute exactly once; it either succeeds or fails. Transactional RPC uses the **exactly-once** semantic.

Communications Failures

If a server detects a communications failure during a remote procedure call, the server runtime attempts to terminate the now orphaned call by sending a cancel to the called procedure. A *cancel* is a mechanism by which a client thread of execution notifies a server thread of execution (the canceled thread) to terminate as soon as possible. A cancel sent by the RPC runtime after a communications failure initiates orderly termination for a remote procedure call.

Clients Contacting Servers

Endpoints and the Endpoint Mapper Service

There are two parts to a server's location: the network address of the system it resides on and the transport-specific address of a specific server instance on that system, that is, the endpoint. Two kinds of endpoints exist: well-known and dynamic endpoints. A well-known endpoint is a preassigned stable address that a server can use every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol. For example, the ARPANET Network Information Center assigns endpoint values for the IP family of protocols. Dynamic endpoints are requested and assigned at runtime.

RPC uses a specialized type of directory service to store endpoints called the Endpoint Mapper Service. When a server starts, it registers its endpoint(s) to make them available to clients. Most servers register an endpoint for each transport protocol supported on the server (for example, TCP and UDP).

Every system that runs an RPC server also runs the Endpoint Mapper Service. This service always uses the same endpoint so its process address is well-known. Therefore, once a client knows what system a server is running on, it can find the Endpoint Mapper running on that same system. It can then ask the Endpoint Mapper for the endpoint of the server process.

Binding Information and Binding

Binding information includes a set of information that identifies a server to a client or a client to a server.

- **A protocol sequence**: A valid combination of communications protocols. An RPC server tells the runtime which protocol sequences to use when listening for calls to the server, and its binding information contains those protocol sequences.
- **Network addressing information**: Includes the network address and the server endpoint.
- **Transfer Syntax**: The server's RPC runtime must use a transfer syntax that matches one used by the client's RPC runtime. A transfer syntax is a set of encoding rules used for the network transmission of data and the conversion to and from different local data representations. ASN.1 is an example of a transfer syntax.
- **RPC protocol version numbers**: The client and server runtimes must use compatible versions of the RPC protocol specified by the client in the protocol sequence.

Compatible binding information identifies a server whose communications capabilities (RPC protocol and protocol major version number, network and transport protocols, and transfer syntax) are compatible with those of the client. Compatible binding information is sufficient for establishing a binding. However, binding information is insufficient for ensuring that the binding is to a compatible server, that is, a server that also offers the requested RPC interface and RPC object (if any). The server may not exist or support the expected RPC interface or RPC object.

In order for the client to send an RPC to the server, it must be able to find the server. This process is called binding. A client may have a direct way of knowing what server it needs to communicate with by obtaining this information from a file, a value hardcoded into its program, an environment variable, or a user. But, a more flexible way for a client to find a server is to take advantage of RPC's use of the Directory Service.

A client can find a server by asking the Directory Service for the location of a server that handles the interface the client is interested in. A client can delegate the finding of servers from the namespace to a stub. In this case, if a binding is accidentally broken, the RPC runtime automatically tries to establish a new binding with a compatible server. The RPC runtime finds the server process endpoint by contacting the server's Endpoint Mapper Service. In order for the Directory Service to be able to give the client this information, a server must first advertise itself in the Directory Service. The server adds itself to the namespace with information about what interfaces it implements, what protocols it uses to communicate with, and where it is located, the binding information. This way, a server can move, or there can be multiple servers implementing a given interface, without affecting the client. The client can still go to the Directory Service and find out where the server is located.

The client program uses the Name Service Independent (NSI) API, an RPC-specific name service layer, to locate an appropriate server with which to bind in order to execute the remote procedure call. Calls to the NSI API are made by the client side of an application in order to look up binding information for a server based on various criteria, such as the name or type of service, the objects it manages, and the interfaces it supports. The server side of an application calls this library to advertise information about itself to the namespace where clients can find it.

How It Works

Let's walk through a simple example to show how the RPC concepts presented work. A Bank is a client-server application. The server offers an account procedure called *balance*. The client requests the balance of an account.

The balance interface is defined in IDL. The IDL compiler compiles this interface definition to produce a client stub, server stub, and header file. The balance procedure is written, compiled and linked with the bank server application, the server stub. The client application is written, compiled, and linked with the client stub. (See Figure 1 on page 8.)

The bank server part of the application must first initialize itself (as shown in Figure 3 on page 13). The bank server has the following as part of initialization:

1. Registers the interface (remote procedure) it offers with the RPC runtime, the balance procedure.
2. Creates the binding information.
3. Advertises its location with the directory service: the balance interface and its server address.
4. Registers the endpoint with the endpoint mapper server.
5. Listens for calls on that endpoint.

The client application now requests the balance of an account. (See Figure 3 on page 13.)

6. The application calls the remote procedure balance.
7. The stub finds the server for this request by contacting the directory server.
8. The RPC runtime contacts the endpoint mapper service on that server to obtain the endpoint for the balance process.
9. The client binds to the server.

Now the remote procedure executes and the parameters (input and output) are sent between client and server. (See Figure 4 on page 13.)

10. The client stub marshals the input argument (acct) and dispatches the call to the RPC runtime.
11. The client RPC runtime transmits the input argument (acct) over the communications network to the server's RPC runtime which dispatches the call to the server stub for the balance procedure.
12. The server stub unmarshals the input arguments and passes them to the balance procedure.
13. The balance procedure executes and returns results (the balance) to the server's stub.
14. The server stub marshals the result and passes it to the RPC runtime.
15. The server RPC runtime transmits the balance over the communication network to the client's RPC runtime which dispatches it to the balance stub of the client.
16. The client stub unmarshals the return result (balance) and passes it to the calling code.

The mechanisms in both the client and server stubs and the Runtime library are transparent to the application programmer. The programmer simply writes the application calls to the RPC operations on the client side, and provides implementations for those operations on the server side. The network communications code is generated automatically. Note that Step 7 can occur in the client application prior to invoking the balance RPC.

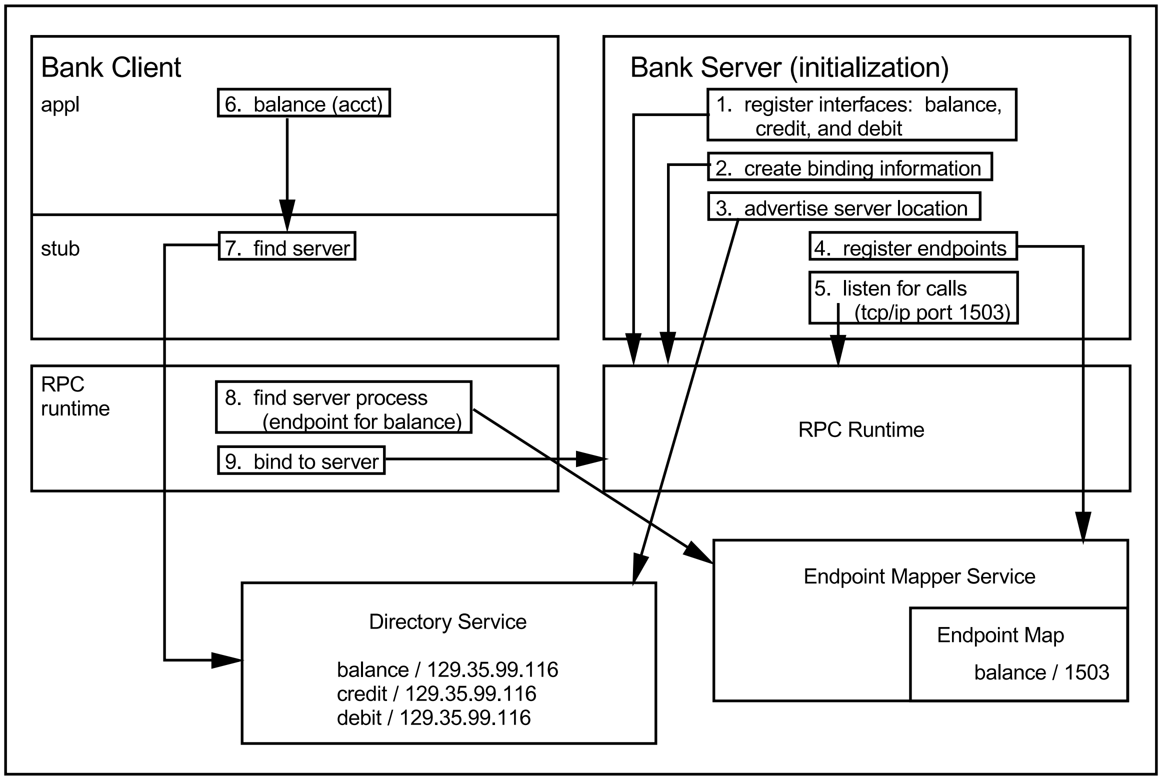


Figure 3. Server Initialization and Client Bind to Server

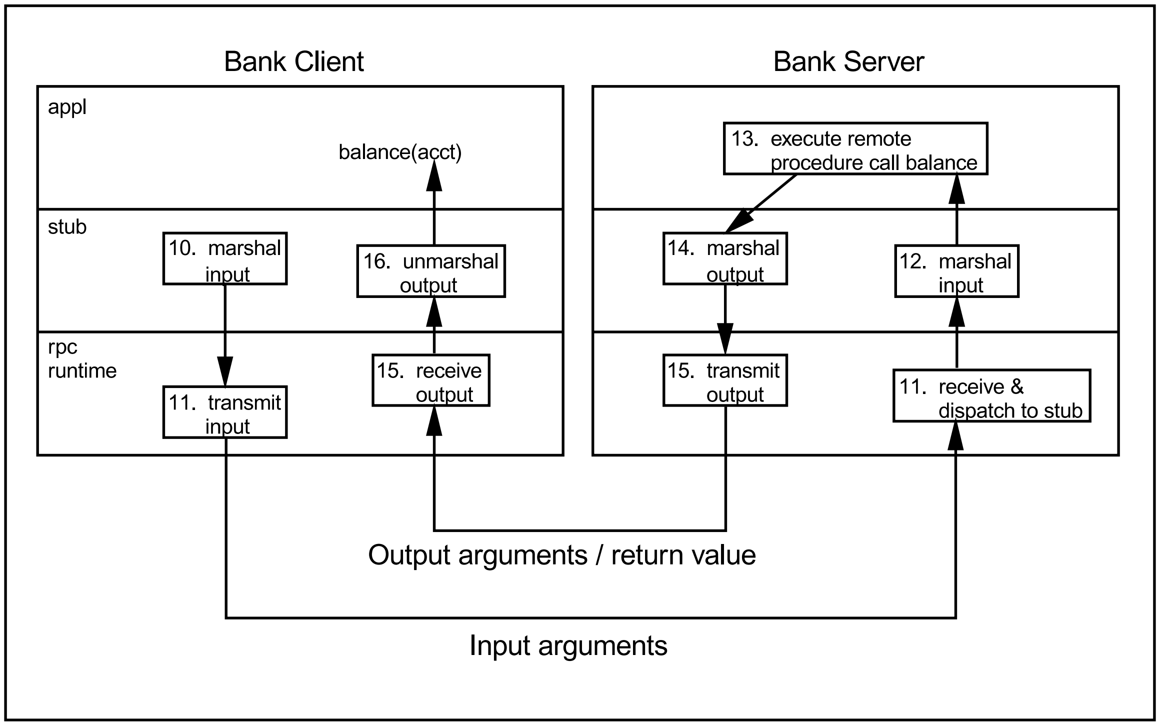


Figure 4. Execution of the Remote Procedure Call

Programming Support

The Interface Description Language (IDL)

The Interface Description Language is a language for specifying operations (procedures or functions), parameters to these operations, and data types. The supported data types are:

- Basic
  - floating point (float and double)
  - integer (signed and unsigned; hyper, long, short, and small)
  - character
  - boolean
  - handle
  - void
- Constructed types
  - structures
  - unions
  - enumerations
  - pipes
  - arrays
  - strings

RPC Runtime API

The RPC API can be classified into the following types of routines:

- Binding operations to manipulate binding information
- Interface operations to manipulate interface information
- Protocol sequence operations: establish and query
- Local endpoint operations to manipulate information in a application's local endpoint map
- Object operations to manipulate object related information, for example, UUID, type UUID mappings
- Name service interface (NSI): binding, entry, group, and profile operations
- Authenticated RPC operations: authentication, authorization, and protection level
- Server listen for servers to begin listening for remote procedure calls
- String free for applications to free string memory allocated by RPC APIs that return strings
- UUID operations to manipulate UUIDs
- Stub memory management to allow applications to participate in memory management
- Endpoint management to manipulate endpoint maps
- Name service management
- Local management services - miscellaneous local operations for servers and clients to manage their RPC interactions
- Local/remote management services to query and stop servers remotely
- Error message operations to provide a locale-independent way to get error message text for a status code returned by an RPC API routine

RPC and Threads

Thread services are also important to RPC. A thread is a single sequential flow of control with one point of execution on a single processor at any instant. Multiple threads can coexist in a single process. RPC uses threads internally for its own operations. RPC also provides an environment where its applications can use thread services. A thread created and managed by application code is termed an “application thread.”

Traditional processing occurs exclusively within local application threads. Local application threads execute within the confines of one address space on a local system and pass control exclusively among local code segments.

RPC applications use application threads to issue both remote procedure calls and runtime calls, as follows:

- An RPC client contains one or more client application threads; that is, a thread that executes client application code that makes one or more remote procedure calls.
- An RPC server uses one server application thread to execute the server application code that listens for incoming calls.

In addition, an RPC server uses one or more call threads that the RPC runtime provides for executing called remote procedures. As part of initiating listening, the server application thread specifies the maximum number of concurrent calls it will execute. The maximum number of call threads in multi-threaded applications depends on the design of the application. The RPC runtime creates the same number of call threads in the server process.

The number of call threads is significant to application code. Application code does not have to protect itself against concurrent resource use when using only one call execution thread, but must when using more than one call thread.

Remote Procedure Call Threads

In distributed processing, a call extends to and from client and server address spaces. Therefore, when a client application thread calls a remote procedure, it becomes part of a logical thread of execution known as an RPC thread. An RPC thread is a logical construct that encompasses the various phases of a remote procedure call as it extends across actual threads of execution and the network. After making a remote procedure call, the calling client application thread becomes part of the RPC thread. Usually, the RPC thread maintains execution control until the call returns.

An RPC server can concurrently execute as many remote procedure calls as it has call threads. When a server is using all of its call threads, the server application thread continues listening for incoming remote procedure calls. While waiting for a call thread to become available, RPC server runtimes can queue incoming calls. Queuing incoming calls avoids remote procedure calls failing during short-term congestion.

Cancels

RPC uses and supports the synchronous cancel capability provided by POSIX threads (pthreads). A cancel is a mechanism by which a thread informs another thread to terminate as soon as possible. Cancels operate on the RPC thread exactly as they would on a local thread, except for an application-specified, cancel-time-out period. A cancel-time-out period is an optional value that limits the amount of time the canceled RPC thread has before it releases control.

During a remote procedure call, if its thread is canceled and the cancel-time-out period expires before the call returns, the calling thread regains control and the call is orphaned at the server. An orphaned call may continue to execute in the call thread, however it is no longer part of the RPC thread, and the orphaned call will be unable to return results to the client.

A client application thread can cancel any other client application thread in the same process (it is possible, but unlikely, for a thread to cancel itself). While executing as part of an RPC thread, a call thread can be canceled only by a client application thread.

Multi-threaded RPC Applications

RPC provides an environment for RPC applications that create multiple application threads (multi-threaded applications). The threads of a multi-threaded application share a common address space and much of the common environment. If a multi-threaded application must be thread-safe (guarantees that multiple threads can execute simultaneously and correctly), the application is responsible for its own concurrency control. Concurrency control involves programming techniques such as controlling access to code that can share a data structure or other resource to prevent conflicting overlapping access by separate threads.

A multi-threaded RPC application can have diverse activities going on simultaneously. A multi-threaded client can make concurrent remote procedure calls and a multi-threaded server can handle concurrent remote procedure calls. Using multiple threads allows an RPC client or server to support local application threads that continue processing independently of remote procedure calls. Also, multi-threading enables the server application thread and the client application threads of an RPC application to share a single server address space as a joint client/server instance. A multi-threaded RPC application can also create local application threads that are uninvolved in the RPC activity of the application.

Performance Implications

RPC is a fundamental enabler of distributed computing. Unlike local applications, RPC applications require network resources, which are possible bottlenecks to scaling an RPC application. RPC clients and servers require network resources that are not required by local programs. The main network resources to consider are network bandwidth, endpoints, network descriptors (the identifiers of potential network channels such as UNIX sockets), kernel buffers, and for a connection-oriented transport, the connections. Also, RPC applications place extra demands on system resources such as memory buffers, various quotas, and the CPU.

The number of remote procedure calls that a server can support depends on various factors, such as the following:

- The resources of the server and the network
- The requirements of each call
- The number of calls that can be concurrently offered at some level of service
- The performance requirements

An accurate analysis of the requirements of a given server involves detailed work load and resource characterization and modeling techniques. Although measurement of live configurations under load will offer the best information, general guidelines apply. You should consider the connection, buffering, bandwidth, and CPU resources as the most likely RPC bottlenecks to scaling.

Relationships with Other Open Blueprint Distributed Services

RPC is a fully integrated part of the distributed computing environment. The communications capabilities of RPC are used by clients and servers of other Open Blueprint components. In turn, RPC uses services provided by the Common Transport Semantics, the Security Service, and the Directory Service of the Open Blueprint.

The RPC runtime uses the Open Blueprint Common Transport Semantics to insulate itself from the underlying transport network.

The RPC runtime provides RPC applications with a programming interface to the Security Service. The RPC authentication interface enables RPC clients and servers to mutually authenticate (that is, prove the identity of) each other. An authenticated remote procedure call provides client authentication information and authorization information to servers. Client authentication information identifies a client to a server. Authorization information includes the privileges a client has and the identities a client is associated with at the time of a call. By comparing client authorization information to access control lists, a server can find out whether a client is eligible to use a requested remote procedure.

To help RPC clients find RPC servers, RPC applications typically use a namespace. A namespace is a collection of information about applications, systems, and any other relevant computing resources. A namespace is maintained by a directory service such as the Cell Directory Service (CDS). RPC provides a Name Service Interface (NSI) that is independent of any particular directory service.

NSI communicates with supported directory services for both RPC applications and the RPC control program. NSI insulates RPC applications from the intricacies of using a directory service. An RPC server uses NSI to store information about itself in a namespace, and a client uses NSI to access information about a server that meets the client's requirements for a specific RPC interface and object, as well as other criteria. The client uses this information to establish a relationship, known as a binding, with the server.

Common Transport Semantics (CTS) and RPC

For an RPC client and server to communicate, their RPC runtimes must use a common RPC protocol. An RPC protocol is a communications protocol that supports the semantics of the RPC API. RPC provides two RPC protocols, the connectionless RPC protocol and the connection-oriented RPC protocol. These RPC protocols run over a corresponding CTS/Transport Network.

- Connectionless (Datagram) RPC protocol

  This protocol runs over a CTS connectionless transport protocol such as UDP. The connectionless protocol supports broadcast calls.

- Connection-oriented RPC protocol

  This protocol runs over a CTS connection-oriented transport protocol such as TCP.

Security and RPC

RPC supports authenticated communications between clients and servers. Authenticated RPC uses the authentication and authorization service provided by the Open Blueprint Security resource manager.

On the application level, a server makes itself available for authenticated communications by registering its principal name and the quality of protection attributes that it supports with the RPC runtime. The server principal name is the name used to identify the server as a principal to the registry service. In practice, this name is usually the same name that the server uses to register itself with the Directory Service.

A client must establish its identity as well as appropriate quality of protection attributes that it wishes to use in its communications with a server. The client identifies the intended server by means of the principal name that the server has registered with the RPC runtime. Once the required quality of protection attributes have been established for the server binding handle, the client issues remote procedure calls to the server as it does in unauthenticated communication.

The Open Blueprint Security resource manager, in conjunction with the RPC runtime, assumes responsibility for the following:

- Authenticating the client and server using the authentication service specified by the quality of protection attributes
- Applying the level of protection specified by the quality of protection attributes to communications between the client and server
- Providing client authorization data to the server in a form determined by the requested authorization service

Authentication

Before a client and server can engage in authenticated RPC, they must ‘agree’ on which authentication service to use. The server must register the ‘agreed on’ authentication service with the RPC runtime along with the server's principal name. The client must select the same service for that server. The different authentication services that can be requested are:

- No authentication
- Shared-secret key authentication (DCE Kerberos version 5)
- Public key authentication
- Default authentication service (chosen from among the previous choices)

A client can engage in authenticated RPC with a target server that is in the client's cell or in a foreign cell.

When a client establishes authenticated RPC, it can specify the quality of protection to be applied to its communications with the server. The quality of protection attributes determine how client/server messages are encrypted. As a rule, the stronger the protection level, the greater the impact on performance. Different levels are provided so that applications can control the protection against performance trade-offs.

The quality of protection can be set as follows:

- Default - the default protection level for the specified authentication service
- None - no protection
- Connect - authentication performed only when the client establishes a relationship with the server
- Call - authentication performed at the beginning of each remote procedure call when the server receives the request

- Packet - authentication data is sent with each packet transmitted from client to server
- Packet integrity - ensures and verifies that none of the data transferred between client and server has been modified; uses cryptographic checksum
- Packet privacy - encrypts all client/server message data (the strength of protection available outside U.S.A. varies by vendor because of cryptographic export regulations)
- CDMF - IBM-only packet privacy option using 40-bit cryptographic keys

Access Control

Access Control is the process of checking the validity of a client's request to perform an operation on an object based on the client-provided credentials.

When a client establishes authenticated RPC, it must indicate which authorization option it wants to use to make client authorization information available to servers for access checking. Authenticated RPC supports the following options:

- None - no authorization information is provided and therefore the server will reject the request if its policy requires authorization information.
- Name - the client principal name is provided to the server; the server performs checking based on name only.
- DCE - additional group names and attributes are provided by the client as part of the authentication credentials for the server to make the appropriate access checks.

Authenticated RPC

Authenticated RPC is implemented as a set of related RPC routines. Some of the routines are for use by clients, some by servers and their managers, and some by both clients and servers. These routines establish, register, inquire, set authentication service, protection level, and authorization service.

Directory Services and RPC

The RPC Name Service Interface (NSI) configures directory service entries and is used by RPC applications to access those entries.

NSI Directory Service Entries

To store information about RPC servers, interfaces, and objects, NSI defines the server entries, groups, and profiles in the namespace. These directory service entries are Cell Directory Service objects.

- A server entry is a directory service entry that stores binding information and object UUIDs for an RPC server.
- A group is a directory service entry that corresponds to one or more RPC servers that offer one or more RPC interfaces, type of RPC object, or both in common.
- A profile is a directory service entry that defines search paths in a namespace for a server that offers a particular RPC interface and object.

The use of server entries, groups, and profiles determines how clients view servers. A server describes itself to its clients by exporting binding information associated with interfaces and objects to one or more server entries. A group corresponds to servers that offer a given interface, service, or object. Profiles enable clients to access alternative directory service entries when searching for an interface or object.

Used together, groups and profiles offer sophisticated ways for RPC applications to maintain and use directory service data.

NSI Attributes

The way NSI stores RPC information allows you to store server entries, groups, and profiles in a directory service entry. To store information about RPC applications in a directory service entry, the RPC directory service interface defines several RPC-specific directory service attributes, or NSI attributes. NSI attributes contain information about RPC applications in a directory service entry. The NSI attributes are as follows:

- NSI binding attribute

  Stores binding information and interface identifiers (interface UUID and version numbers). This attribute identifies a directory service entry as a server entry.

- NSI object attribute

  Stores a list of one or more object UUIDs. Whenever a server exports any object UUIDs to a server entry, the server entry contains an object attribute as well as a binding attribute. When a client imports from that entry, the import operation returns an object UUID from the list stored in the object attribute.

- NSI group attribute

  Stores the entry names of the members of a single group. This attribute identifies a directory service entry as an RPC group.

- NSI profile attribute

  Stores a set of profile elements. This attribute identifies a directory service entry as an RPC profile.

Any directory service entry can contain any combination of the four NSI attributes. In addition to the NSI attributes, a directory service entry contains other kinds of attributes. Every entry in a namespace contains standard attributes created by the directory service. NSI operations rely on some standard attributes to identify and use an entry. Any directory service entry can also contain additional attributes specified by non-RPC applications; these are ignored by NSI operations.

Server Entries

NSI enables any RPC server with the necessary directory service permissions to create and maintain its own server entries in a namespace. A server can use as many server entries as it needs to advertise combinations of its RPC interfaces and objects.

Each server entry corresponds to a single server (or a group of interchangeable server instances) on a given system. Interchangeable server instances are instances of the same server running on the same system that offer the same RPC objects (if any). Only interchangeable server instances can share a server entry.

Each server entry contains binding information. Every combination of protocol sequence and network addressing information represents a potential binding. The network addressing information can contain a network address, but lacks an endpoint, making the address partially bound. A server entry can also contain a list of object UUIDs exported by the server. Each of the object UUIDs corresponds to an object offered by the server. In a given server entry, each interface identifier is associated with every object UUID, but only with the binding information exported with the interface.

RPC Groups

Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC group as the starting point for NSI search operations. A group provides NSI search operations with access to the server entries of different servers that offer a common RPC interface or object. A group contains names of one or more server entries, other groups, or both. Since a group can contain group names, groups can be nested. Each server entry or group named in a group is a member of the group. A group's members offers one or more RPC interfaces, the type of RPC object, or both in common.

RPC Profiles

Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC profile as the starting point for NSI search operations. A profile is an entry in a namespace that contains a collection of profile elements. A profile element is a directory entry that corresponds to a single RPC interface and that refers to a server entry, group, or profile. Each profile element contains an interface identifier, member name, priority value, and annotation string. Optionally, a profile can contain one default profile element.

Searching the Namespace for Binding Information

Searching the namespace for binding information requires that a client specify a starting point for the search. A client can start with a specific server entry. However, this is a limiting approach because the client is restricted to using one server. To avoid this, a client can start searching with a group or a profile instead of with a server entry. The NSI search operations traverse one or more entries in the namespace when searching for compatible binding information. In each directory service entry, these operations ignore non-RPC attributes and process the NSI attributes in the following order:

- Binding attribute (and object attribute, if present)
- Group attribute
- Profile attribute

If an NSI search path includes a group attribute, the search path can encompass every entry named as a group member. If a search path includes a profile attribute, the search path can encompass every entry named as the member of a profile element that contains the target interface identifier. A search finishes only when it finds a server entry containing compatible binding information and the non-null object UUID, if requested.

The Models for Defining Servers

The NSI operations accommodate two distinct models for defining servers, the service model and the Resource model. These models express different views of how clients use servers and how servers can present themselves in the directory service database. The models are not mutually exclusive, and an application may need to implement both models to meet diverse goals. Evaluation of these models before designing an RPC application can help answer questions about whether and how to use object UUIDs, how many server entries to use per server, how to distinguish among instances of a server on a system, whether and how to use groups or profiles or both, and so forth. The two models used are the Service model and the Resource model.

The Service model views a server exclusively as a distributed service composed of one or more application-defined interfaces that meet a common goal independently of specific resources. The Service model is used by applications whose servers offer an identical service and whose clients do not request an RPC resource when importing an interface. Often, with the Service model, all the server instances of an application are equivalent and are viewed as interchangeable. However, the Service model can accommodate applications that view each server instance as unique.

The Resource model views servers and clients as manipulating resources. A server and its clients use object UUIDs to identify specific resources. With the Resource model, any resource an application's servers and clients manipulate using an object UUID is considered an RPC resource. Typically, an RPC resource is a physical resource such as a database. However, an RPC resource may be abstract; for example, a print format such as ASCII. An application that uses the Resource model for one context may use the service model for another. Each RPC resource or type of resource requires its own object UUID. Some RPC resources belong exclusively to a single server instance; other RPC resources can be shared among server instances. Multiple server instances that access the same set of resources can introduce concurrency control problems, such as two instances accessing a tape drive at the same time. Also, where the system provides concurrency control, servers may compete and have to wait for resources such as databases. Dealing with delayed access to shared resources may require an application-specific mechanism, such as queuing access requests.

RPC Administration

A few administrative tasks must be performed when running a distributed application using RPC. The application server executes most of these tasks when it first starts. The server registers its (dynamically assigned) listening endpoint with the Endpoint Mapper Service. The server also advertises information about itself and the interfaces it supports in the Directory Service.

Non-automated RPC administration is minimal. Each distributed system has a host daemon running on it. An administrator may be involved in registering servers in the namespace, but this can also be done from server code upon initialization as just described. Usually, an administrator will be needed to change the ACL on the directory where the server will register so that the server has write permission. A management program is provided to allow an administrator to (among many things) control the Host daemon and administer RPC information in the namespace.

An administrator may be involved in installing a new RPC-based application. In particular, the server side of the application must be started before it can begin receiving and servicing requests. The administrator may arrange for the server process to be started each time the system is booted, for example.

A graphical user interface is provided for RPC Administration.

Appendix. Internationalized RPC Applications

An internationalized RPC application is one that:

- Uses the operating system platform's locale definition functions to establish language-specific and culture-specific conventions for the user and programming environment
- Isolates all user-visible messages into message catalogs
- Uses RPC-provided or user-defined character and code set evaluation and automatic conversion features to ensure character and code set interoperability during the transfer of international characters in remote procedure calls between RPC clients and servers.

A *locale* defines the subset of a user's environment that depends upon language and cultural conventions. A locale consists of categories; each category controls specific aspects of some operating system components' behaviors. Categories exist for character classification and case conversion, collation order, date and time formats, numeric non-monetary formatting, monetary formatting, and formats of informative and diagnostic messages and interactive responses. The locale also determines the character sets and code sets used in the environment. The syntax and use of a locale definition function depends on the operating system platform in use.

A character set is a group of characters, such as the English alphabet, Japanese Kanji, and the European character set. To enable world-wide connectivity, DCE guarantees that a minimum group of characters is supported in the DCE. RPC communications protocol ensures this guarantee by requiring that all RPC clients and servers support the DCE Portable Character Set (PCS).

A ‘code set’ is a mapping of the members of a character set to specific numeric code values. Examples of code sets include ASCII, JIS X0208 (Japanese Kanji), and ISO 8859-1 (Latin 1). The same character set can be encoded in different code sets; consequently, a distributed environment can contain RPC clients and servers that use the same character set but represent that character set in different numeric encodings. “Code set conversion” is the ability for an RPC client or server to convert character data between different code sets.

The RPC communications protocol, through the Network Data Representation (NDR) transfer syntax, provides automatic code set conversion for DCE PCS characters encoded in two code sets: ASCII and EBCDIC 500. The RPC communications protocol automatically converts character data declared as char or idl_char between ASCII and EBCDIC 500 encodings, as necessary, for all RPC clients and servers.

The RPC communications protocol does not provide support for the recognition of characters outside of the DCE PCS, nor does it provide automatic conversion for characters encoded in code sets other than ASCII and EBCDIC 500.

However, RPC does provide IDL constructs and RPC runtime routines that programs can use to exchange non-PCS, or international character data that is encoded in code sets other than ASCII and EBCDIC 500. These features provide mechanisms for international character and code set evaluation and automatic code set conversion between RPC clients and servers. Using these features, programs can run in a distributed environment that supports multiple heterogeneous character sets and code sets.

There is a code set registry (a per-system file) that contains mappings between string names for the supported code sets and their unique identifiers. OSF assigns the unique identifiers for the code sets and administrators assign their platform string names for the code sets. The RPC routines for character set and code set interoperability depend upon a code set registry existing on each system.

Remote Procedure Call with Character/Code Set Interoperability

The basic tasks of an RPC application (client, client stub, server, and server stub) as described earlier are integrated with the additional tasks (highlighted) required to implement an RPC that provides character and code set interoperability.

Figure 5 (Page 1 of 2). Tasks of an Internationalized RPC Application

Client Tasks	Server Tasks
	1. **Set locale**
	2. Select network protocols
	3. Register RPC interfaces
	4. Advertise RPC interfaces and objects in the namespace
	5. **Get supported code sets and register them in the namespace**
	6. Listen for calls
7. **Set locale**	
8. **Establish character and code sets evaluation routine**	
9. Find compatible servers that offer the procedures	
10. Call the remote procedure	
11. Establish a binding relationship with the server	
12. **Get code set tags from binding handle**	
13. **Calculate buffer size for possible conversion of input arguments from local to network code set**	
14. **Convert input arguments from local to network code set (if necessary)**	
15. Marshal input arguments	
16. Transmit arguments to the server's runtime	
	17. Receive call
	18. Get code set tags sent from client
	19. **Calculate buffer size for possible conversion of input arguments from network to local code set**
	20. Unmarshal input arguments
	21. **Convert input arguments from network to local code set (if necessary)**
	22. Locate and invoke the called procedure
	23. Execute the remote procedure
	24. **Calculate buffer size for possible conversion of output arguments from local to network code set**
	25. **Convert output arguments from local to network code set (if necessary)**
	26. Marshal output arguments and return value
	27. Transmit results to the client's runtime
	28. **Remove code set information from namespace on exit**

Figure 5 (Page 2 of 2). Tasks of an Internationalized RPC Application

Client Tasks	Server Tasks
29. Receive results	
30. **Calculate buffer size for possible conversion of output arguments from network to local code set**	
31. Unmarshal output arguments	
32. **Convert output arguments from network to local code set (if necessary)**	
33. Pass to the calling code the results and return control to it	

The stub conversion routines do not implement code set conversion. Instead, they call POSIX-compliant **iconv** code set conversion routines, which are part of a local operating system.

The Evaluation Routine

When a client attempts to import binding information from a namespace entry that it looks up by name, the NSI import routine checks the binding for compatibility with the client by comparing interface UUIDs and protocol sequences. Internationalized clients need to perform additional compatibility checking on potential server bindings—they need to evaluate the server for character and code set compatibility.

The purpose of the character and code sets compatibility evaluation routine is to determine:

- Whether the character set the server supports is compatible with the client's character set, since incompatible character sets result in unacceptable data loss during character conversion.
- The level of code sets compatibility between client and server, which determines the conversion method that the client and server will use when transferring character data between them.

A conversion method is a process for converting one code set into another. There are four conversion methods:

- Receiver Makes It Right (RMIR)—the recipient of the data is responsible for converting the data from the sender's code set to its own code set. This is the method that the RPC communications protocol uses to convert PCS character data between ASCII and EBCDIC code sets.
- Client Makes It Right (CMIR)—the client converts the input character data to be sent to the server into the server's code set before the data is transmitted over the network, and converts output data received from the server from the server's code set into its local code set.
- Server Makes It Right (SMIR)—the server converts the input character data received from the client into its local code set from the client's code set and converts output data to be sent to the client into the client's code set before the data is transmitted over the network.
- Intermediate—Both client and server convert to a common code set. DCE defines a default intermediate code set to be used when there is no match between the client and server's supported code sets; this code set is the ISO 10646 universal code set. Other code sets can be specified to be used as intermediate code sets in preference to ISO 10646.

A character and code sets compatibility evaluation routine generally employs a conversion model when determining the level of code sets compatibility. A conversion model is an ordering of conversion methods, for example, ‘CMIR first, then SMIR, then intermediate.’ The actual conversion method used is determined at runtime.

RPC Evaluation Routines

RPC provides two character and code sets compatibility evaluation routines:

- With universal
- Without universal

If client and server are using the same code set, then have client-server character and code set compatibility to exchange data; no further evaluation for compatibility occurs and no conversion during transmission of data occurs. Otherwise, the evaluation routine determines what conversion method to use. If the client and server are using the same character set, it will be safe for them to exchange character data despite their use of different encodings for the character data. Clients and servers using different character sets are considered to be incompatible, since the process of converting the character data from one character set to the other will result in significant data loss.

The client and server's local code set identifiers are used to obtain (from the Code Set Registry) the registered values that represent the character set(s) that the specified code sets support to determine compatibility.

In the case where the client and server are character set compatible, the 'with universal' uses the following model to determine a conversion method:

- RMIR (receiver makes it right)
- SMIR (client uses its local code set, server converts to and from it)
- CMIR (server uses its local code set, client converts to and from it)
- Use an intermediate code set
- Use the universal (ISO 10646) code set

This conversion model translates into the following steps:

- If both client and server support converters for each others local code sets, that is, they can convert to and from each others local code set, the routine sets the conversion method to RMIR.
- If the server can convert to and from the client's local code set, but the client cannot convert from the server's local code set, the routine sets the conversion method to SMIR.
- If the client can convert to and from the server's local code set, but the server cannot convert to and from the client's local code set, the routine sets the conversion method to CMIR.
- If neither client nor server support each other's local code set, the routine next determines if they both support a code set into which they both can convert to from their local code sets. If it finds an intermediate set into which they both can convert, it sets the conversion method to **intermediate** and sets the sending tag and desired receiving tag to the code set value that represents the intermediate code set to use.
- If the routine does not find any intermediate code set into which client and server can convert, it sets the sending tag and desired receiving tag to the code set value that represents the ISO 10646 universal code set, which is the default intermediate code set that all DCE clients and servers support.

The 'without universal' evaluation routine uses the following conversion model to determine a conversion method:

- RMIR
- SMIR (client uses its local, server converts to and from it)
- CMIR (server uses its local, client converts to and from it)
- Intermediate
- Reject for code set incompatibility

Appendix B. Bibliography

OSF DCE Application Development Reference Version 1.1

OSF DCE Application Development Guide Version 1.1

OSF DCE Administration Guide Version 1.1

OSF DCE Command Reference Version 1.1

Introduction to OSF DCE Version 1.1

OSF DCE RFC 41.1 “RPC Runtime Support for I18N Characters”

X/Open, *DCE: Remote Procedure Call*, Document Number C309, ISBN 1-85912-041-5

X/Open, *DCE: Directory Services*, Document Number C312, ISBN 1-85912-078-4

Understanding DCE by Ward Rosenberry, David Kenney, & Gerry Fisher, ISBN 1-56592-005-8

Guide to Writing DCE Applications by John Shirley, Wei Hu, & David Magid, ISBN 1-56592-045-7

Open Blueprint Transaction Manager Resource Manager Paper

Encina Transactional-C Programmer's Guide and Reference

Appendix C. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM
IBMLink
Open Blueprint

The following terms are trademarks of other companies:

DCE	Open Software Foundation, Incoporated
Encina	Transarc Corporation
OSF	Open Software Foundation, Incoporated
POSIX	Institute of Electrical and Electronic Engineers
X/Open	X/Open Company Limited

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Appendix D. Communicating Your Comments to IBM

If you especially like or dislike anything about this paper, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply. Feel free to comment on specific error or omissions, accuracy, organization, subject matter, or completeness of this paper.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send comments by FAX, use this number:

  United States and Canada: 1-800-227-5088.

- If you prefer to send comments electronically, use one of these ID's:
  - Internet: **USIB2HPD@VNET.IBM.COM**
  - IBM Mail Exchange: **USIB2HPD at IBMMAIL**
  - IBMLink: **CIBMORCF at RALVM13**

Make sure to include the following in your note:

- Title of this paper
- Page number or topic to which your comment applies

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC23-3926-01