REMOTE PROCEDURE CALLS

The remote procedure call (RPC) is a computer technology for interprocess communications. A calling program uses an RPC to invoke a program running in another process. This other process may be running on the local host machine or may be on another host machine. RPC implementations make an interprocess call look similar to a local procedure call, shielding the programmer from many of the complexities of the underlying communication technologies. The RPC technology allows a programmer to code the calling program the same way, regardless of whether the called program is running on the same host or a remote host. Thus, RPCs simplify and standardize interprocess communications. In addition, RPC implementations provide features beyond simple data transmission, including platform independence, location transparency, security, and transport independence, and run on a wide variety of commercial computer platforms.

This article is organized as follows: The first section introduces RPC technology; the following section covers programming with RPCs and generic RPC functions that are common to most RPC implementations; and the last section describes advanced RPC topics.

Examples in this article come from two popular RPC implementations: the SUN ONC+RPC and the Open Software Foundations Distributed Computing Environment (OSF DCE) RPC. Details of how each of these mechanisms implement RPC functionality are discussed in some of the sections.

RPC FEATURES

The main features of RPC programming are as follows:

- *Platform independence.* RPCs compensate for differences in how host platforms represent data internally. All RPC data communication uses standardized data representation. This means that computers having dissimilar internal data representation can communicate properly because all shared data are converted to a standard representation before being sent to the receiver.
- *Location transparency.* Computers that make up the application are not required to be geographically colocated. This means that distributed applications could be spread across large geographic regions to improve response time and availability.
- *Secure network communications.* All commercial computer applications are exposed to security threats. Because the physical communications network is not 100% secure, distributed applications have a higher exposure to security violations. Most RPC mechanisms provide tools to enable various levels of security to guarantee that an application receives the appropriate level of security.
- *Transport independence.* Developers of RPC-based applications do not have to program for a specific underlying network protocol. This means that the communications protocol can be changed at runtime without changes to the program.

RPC TECHNICAL OVERVIEW

The RPC is modeled after the ordinary local procedure call. The differences between the RPC and the local call are extensions resulting from the effects of remoteness that are introduced when the calling and called operation do not share the same process.

Local procedure calls are calls that one part of an active process makes to another part of the same process (or address space). Local procedure calls are also called functions, subroutines, and application programming interfaces (APIs). Like a local procedure call, an RPC may or may not contain input parameters to set initial conditions within the function and output parameters to pass back final state information. Unlike a local procedure call, however, the RPC executes in two separate processes that may or may not be running on the same host. The process that initiates an RPC is called the client and the process that receives, processes, and returns the request is called the server.

From the programmer's point of view, RPCs are rooted in the same programming semantics as local procedure calls. They have well-defined input and output, and the call returns when the function has completed processing (synchronous calls). Having identical semantics as the local procedure call means that the programming with RPCs is virtually identical to programming with the local procedure call. This means that there is very little additional knowledge that a developer must have when working with RPCs. However, as discussed later, there are some additional steps required to check for the effects of remoteness that must be built into the software, especially the consideration of RPC call semantics, which govern how applications behave in the case of errors, such as network failures and server crashes. Figure 1 illustrates the

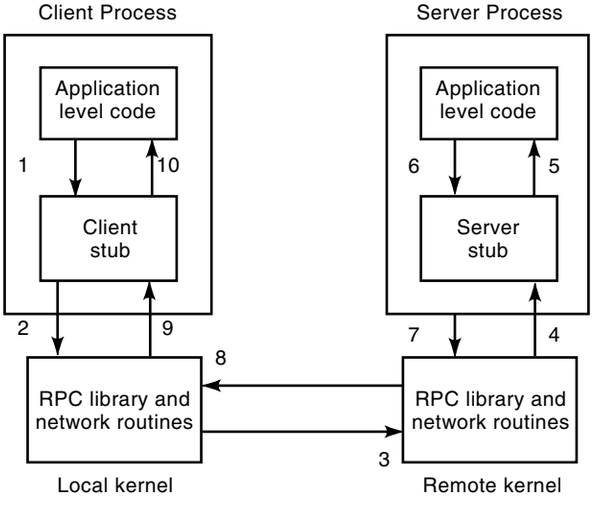

Figure 1. Remote procedure call.

RPC model (1) and demonstrates how all the associated components interact.

1. The client calls the RPC by executing a local procedure call as defined in the interface definition. (The local procedure call is actually a routine in the client stub generated by the interface definition language compiler.)
2. The client stub takes the arguments, if any, and packages them into a standard data representation and forwards the data to the local kernel code for network transmission.
3. The kernel software transmits the information across the network and then waits for the server to reply. This software also executes all actions necessary to support the network protocol used in the call.
4. The information is received by the remote host kernel level code and forwarded to the server stub for processing.
5. The server stub translates the message from the standard data representation into the format described in the interface definition and is passed to the server using a local procedure call.
6. The server processes the request and returns the data to the server stub in the format described in the interface definition. From the server's perspective, it is returning to a local procedure call.
7. The server stub formats the results into a standard data representation format and sends the data back to the kernel code.
8. The kernel code transmits the data back to the client host.
9. The client's kernel receives the data and sends them back to the client's stub.
10. The client stub reformats the data from the standard data representation into the local data representation and returns the data in the format defined in the interface definition.

RPC-BASED DISTRIBUTED COMPUTING BENEFITS AND DRAWBACKS

RPC technology is a basic tool of distributed computing, with attendant benefits and drawbacks.

Benefits of RPC-Based Distributed Computing

Applications developed using RPC-based service providers (servers) enjoy many benefits, including, but not limited to, the following:

- Reduced administration costs in the areas of data backup and recovery. Examples of this class of servers include file and database servers that centralize system and application administration to a single host. Client hosts request files and/or data, manipulate the information, and then store the information back on the server. There can be many clients and a single server. Costs are reduced in two ways. First, client hardware requirements are reduced. Data are centrally stored on the server, so client hosts need not be able to store data. Second, administration costs are reduced. Only the servers, and not each client, are backed up on a regular basis. These cost reductions become more and more significant as the ratio of clients to servers grows.
- Increased application power. As more users store their results in a common store, increasingly powerful applications become possible.
- Elimination of data redundancy problems. Data redundancy occurs when each host requires identical consistent data for the application to run correctly. Managing data consistency between more than one computer is expensive in terms of application design, implementation, and runtime resources. Applications that require common data that are asynchronously accessed are simpler to design and build using the client/server model. This is because there is only one copy of the data, and consistency is not an issue. An example of this type of application would be consistent times among many applications. Keeping multiple hardware clocks at the same time is a complex task because of different clock drift rates and frequency of clock synchronization. However, a time service provider could parcel out time stamps required by clients. Removing the requirement for synchronized clocks greatly simplifies application design, implementation, and administration.
- Reduced specialized hardware requirements. For example, clients running on inexpensive slow computers can send specialized computer-intensive operations to a specialized server that performs mathematical operations very quickly. This promotes the efficient use of specialized equipment.
- Increased availability due to multiple identical servers. Some services can be provided by multiple servers. Availability is increased because the client has multiple servers that it can select from to execute a request. When one server is unavailable, the client can select another server to satisfy its request.

Drawbacks of RPC-Based Distributed Computing

There are design trade-offs in all aspects of computing, and RPC-based processing is no exception. Most of the drawbacks of RPC applications center on the effects of remoteness. Effects of remoteness are defined as the potential for dysfunction because of the following:

- Dependency on the server host being available. Unlike the local procedure call, a client process has no guarantee that the computer running the server it wants to contact is available.
- Dependency on the server process running at the time the client issues the request. In this case, the target computer is available but the required process is not running.
- Dependency on end-to-end network connectivity. Data communications networks are physical entities and therefore subject to outage because of broken wires and network element (routers, bridges, etc.) failure.
- Latency in processing requests. Because RPCs are more likely to require calls across a network in a fully distributed computing environment, clients will have to wait

longer for a response due to the additional network delay.
- Single point of failure. Once a service is centralized, a failure in that service affects more clients.
- Time shifts due to clock skew and time zone differences.

These drawbacks are not the only negative effects of remoteness, but they are the major challenges when designing, building, and deploying distributed applications. Other factors include debugging in a distributed environment, administration of networked computers, and degraded performance due to network congestion and network configuration.

Despite the drawbacks, distributed computing remains a feasible processing model because of the many advances that address the effects of remote computing. For example, good design techniques can be used to compensate gracefully for lack of server and network availability. Other advances in software design and development, such as threads and threads aware debuggers, minimize the effects of remoteness. These design techniques and advances are covered in detail later in this article.

RPC PRODUCTS

The following two subsections describe basic and advanced features of RPC computing. Where appropriate, they include descriptions of how two popular RPC mechanisms, ONC+ and OSF DCE RPC, implement the feature being described. Architecturally, these two products lie between the application and the operating system and network services. Client applications issue a request for service using RPC functions. These functions, in turn, use the operating system and network services to communicate that request to a server and the results of the remote computation back to the client that made the request. Both RPC mechanisms are available on a wide variety of operating systems and support a variety of network-level protocols.

ONC+ RPC

ONC+ is an RPC implementation developed by Sun Microsystems. It provides core services (2) that enable applications developers to design and build distributed applications that run in a heterogeneous environment. The communication mechanism is the synchronous transport-independent (TI) RPC with support for multithreading to enhance concurrency. ONC+ uses external data representation (XDR) to enable platform independence.

OSF DCE

DCE is a collection of services (3) for the development, use, and maintenance of transparent distributed systems using the client/server architecture. DCE enables application-level interoperability and portability through common APIs among heterogeneous platforms. The communication paradigm supported by DCE is synchronous RPC across address spaces (over various network protocols), with multithreading within an address space for concurrency. Client/server location transparency is provided by a directory service/name server. DCE directory services are provided within an administration domain, called a cell, and among cells using Domain Name Service, lightweight directory access protocol (LDAP), and X.500.

DCE security services are based on private (Kerberos) and public key authentication. The security service includes authentication of servers and clients, support for resource authorization by an application server in providing services to its clients, and various levels of message integrity/encryption (at different levels of computing resources). Two other distributed services of DCE are its Distributed File System (DFS), for accessing files across hosts, and Distributed Time Service (DTS), for synchronizing clocks across hosts.

EVOLUTION OF RPCs

RPCs are an evolutionary step in the development of communications tools for distributed computing. RPC technology builds a program-level protocol upon earlier network-level communications protocols. RPC is often layered atop sockets. In turn, object-level communications protocols, such as DCOM and COBRA, have built on top of RPC technology: DCOM explicitly uses DCE RPC, and COBRA reimplements RPC concepts as internal underlying protocols.

PROGRAMMING WITH RPCs

Application development with RPCs is slightly different from typical application development. In addition to the usual steps of writing a client (calling) program and a server (called) program, the developer creates a service interface definition in one of several languages, each called an interface definition language (IDL). The interface definition is passed to a code generator (compiler) that generates stubs for both the client and server. Stubs are linked into the respective program to provide all the functionality necessary to make the RPC work.

There are four stages to developing a RPC based application:

1. Develop a formal description of the services provided by the server. The description includes the service names and their respective input and output parameters. Definitions are written in the IDL of the RPC package used.
2. Generate stub programs for both the client and server. The RPC package provides an IDL compiler that generates stubs from IDL. The stubs transform a local procedure call into a remote procedure call. That is, when invoked by a client via a local procedure call, the client stub makes an interprocess call to the server stub, which invokes the server via a local procedure call. The stubs contain all the code required to marshal the client's arguments into a well-known format, send the data to the server, and return the respective results to the client.
3. Create server and client application software.
4. Compile and link application server code and server stub and application client code and client stub.

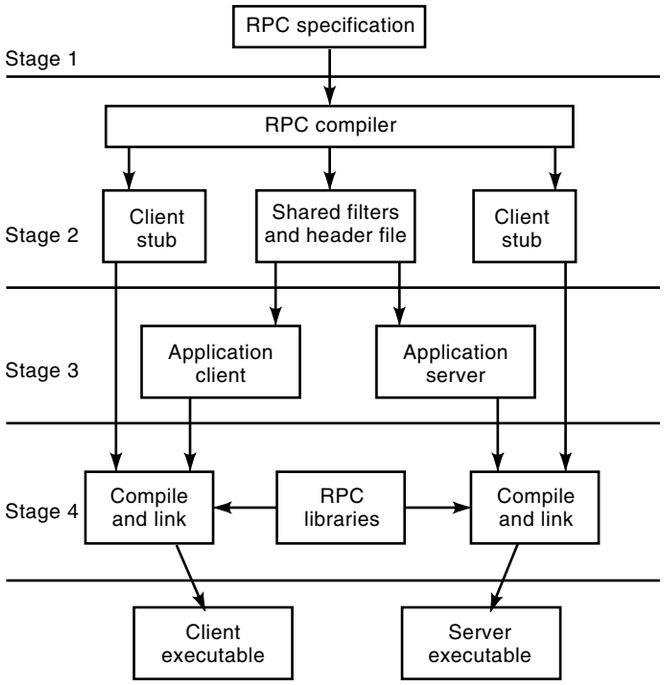

Figure 2. Application integration.

Figure 2 illustrates four stages of building an RPC-based application (4). The shaded boxes indicate steps performed by the applications developer.

Interface Definition

The interface definition is the mechanism by which one describes services offered by a server. The purpose of the interface definition is to define the programming interface between the client and the server. The interface definition is required so that the server knows what the client is passing in as parameters and the client knows what the server is returning.

The interface compiler uses an interface definition to generate the needed header file for the client and server programs as well as stub code for the client to invoke the service and for the server to return the results. The RPC runtime service on each host does data marshaling/unmarshaling necessary for RPC programs to run on different hosts and host types. Marshalling is the process of preparing the parameters to the RPC for transmission across the network. The marshalling logic uses the parameters to the RPC to build a message in a well-known format, which includes data and control information. Unmarshalling is the process of retrieving the parameters to the RPC from the message sent across the network.

The syntax of most IDLs is a C language syntax. Therefore, most interface specifications look like a series of C language declarative statements. Most modern IDLs support primitive data types and allow the developer to create more complex data objects using structures. Also, full pointer support is provided to allow linked lists, nested structures, and reference pointers. Reference pointers can be used to pass parameters whose length is not known until execution time, by roughly effecting call-by-reference semantics.

The ONC+ IDL supports most C language data primitives in the interface definition as well as user-defined data types (5). It also supports multiple input parameters and a single output parameter for each RPC defined. (Previous releases of ONC RPC only supported single input and output parameters, and the output parameter had to be of type static.) RPC arguments cannot be both input and output. Full pointer support means that RPCs are useful for sending recursive data types, such as linked lists and trees, across the network. Multidimensional arrays are defined as linked lists.

DCE RPC IDL supports (6) a wider variety of C language data types than ONC+. It also supports user-defined data types and multiple input and output parameters, and all parameters can be both input and output parameters. Multidimensional arrays can be declared in the language and are not required to be implemented as linked lists. DCE IDL also supports a mechanism called pipes that allows transfer of large quantities of identically typed data.

Integration of Stubs and Application Code. Compiled stubs are used in the application build process to create complete clients and servers. The link step is where this step is performed.

One popular way to develop client/server applications is to build both client and server code together in the same program, debug the application, and then split the program into its respective client and server modules. Although this development strategy is not required, it does help to debug the application before the effects of remoteness are introduced into the application.

RPC Runtime Semantics

Synchronous RPCs. Both RPCs and local procedure calls are synchronous. This means that the process making the call waits for the calling routine to complete its tasks. Because RPCs are made to other processes, they tend to block the client process. Process blocking causes some performance degradation because the client process must wait while the request is transmitted across the network, processed by the server, and transmitted back to the client. The use of threads can alleviate this degradation, as explained later under concurrency.

Asynchronous RPCs. Performance degradation due to blocking RPC calls can be addressed with the use of asynchronous RPCs, also known as one-way RPCs. Asynchronous RPCs are calls that allow the client process to continue processing while the RPC call is pending. When the RPC completes, a predefined callback routine is invoked.

Performance improves because of the increase in concurrency. Concurrency means that the client can perform other operations, including making more RPCs, while waiting for the RPC to complete.

Asynchronous RPCs can be emulated with the use of threads in the client. In the threaded model, the client creates a thread to make the RPC. When the RPC is made, only the thread that made the call blocks. This allows the client process to execute other operations while the RPC is pending (7).

Either practice, asynchronous RPCs or multithreaded programming, alleviates the degraded performance resulting from blocking RPC calls.

Callback RPCs. Callback (8), or follow-up, RPCs are useful when the application cannot wait for the synchronous RPC

to complete and multithreading the client is not an option. Graphical user interface (GUI)–based clients are examples of programs that cannot indefinitely block waiting for a long-running RPC to complete. Essentially, the flow of control in a callback RPC is as follows:

1. The client issues the RPC and blocks waiting for a reply.
2. The server receives the request and returns an acknowledgment to the client to indicate the message was received and starts processing the request.
3. The client receives the server's acknowledgment and continues processing.
4. The server completes the request and sends the output, if any, back to the client.
5. The client receives the reply via application-level polling software.

Both ONC+ and DCE RPC are synchronous calls. ONC+ supports callback routines. Both products support multithreaded programming and therefore allow simulated asynchronous RPCs.

ADVANCED RPC TOPICS

Advanced topics in RPC programming include interface management and change control, platform independence, RPC call semantics, binding information, supporting service providers, concurrency, and event propagation. These subjects are important in discussing any RPC mechanism because they are used by application developers to make better distributed applications. They allow developers to extend the functionality of servers (while not requiring existing clients to change), facilitate location transparency by providing centralized directory services, facilitate secure processing, and increase performance.

Binding Information

Client processes must have information on where a service is being provided to contact, or bind, with the server. Binding information includes the interface number(s), supported network protocols, and network address of the host providing the service. Of these bits of information, the program identification number(s) are usually known at compile time, and the server host location can be obtained by a centralized name service or as a runtime parameter to the client. Network transport selection (e.g., TCP or UDP) is usually negotiated at runtime and may vary with each invocation of the client.

Binding Classes. Server programs listen for requests on a logical device called a port. Port numbers can either be well known (static binding) between the server and client or dynamically assigned (dynamic binding) at runtime. Static bindings assign have their port numbers assigned at compile time, and dynamic binding ports are assigned by a well-known lookup service (well known because the service uses a static binding and resides at the same network address as the server). Client programs consult the lookup service, or port mapper, to obtain the server's port number.

Port mapper services are desirable because the range of well-known transport selectors is very small for some transports and the number of services is potentially very large. By running only the lookup service on a well-known transport selector, the port number of other servers is ascertained by querying the lookup service.

Binding Management. Clients manage dynamic binding information in one of three ways (9). The management technique used dictates the allowable client behavior, and therefore applications must match the correct binding management technique with the expected client behavior.

- Automatic binding is the simplest method because the client stub manages all binding information and completely hides binding information from the application code. Automatic bindings can sometimes automatically retry failed RPCs. Retries are automatically done if the previous RPC never began or when the operation is idempotent, meaning that it can be executed many times without affecting correctness, e.g., a "read" operation. Drawbacks to the automatic binding include not being able to identify server information such as host name and network protocol.
- Implicit binding is slightly more complex than automatic binding because the application must establish the server binding information and assign it to a global variable. The global variable identifies the targeted server in the stub. Benefits of implicit binding include centralized binding assignment. Drawbacks to this method include restrictions on multiple threads using the global variable at the same time.
- Explicit binding is marginally more complex than implicit binding because the application code must explicitly use the binding information on each RPC. Although this method is the most complex, it is also the most flexible. By allowing the client to manage the binding information for individual RPCs, the explicit binding method enable clients to meet special binding requirements.

ONC+ supports explicit binding. The binding information is the last parameter in the RPC call. DCE RPC supports all three types of binding classes. In the case of explicit binding, binding information is the first parameter passed in the RPC call.

Interface Numbers. Interfaces are defined in an IDL and describe the interface specification in human readable form. However, computers require that identification numbers be used to identify services in place of human readable format. To facilitate this mapping of human form to host form, the interface specification must include the interface number and version information as part of the interface definition.

This information is usually passed to the client and server in the common header file generated by the interface compiler. These numbers are used by the servers to register their presence to the local port mapper and by clients to tell the servers port mapper which service they require. The RPC port mapper is usually responsible for routing the initial client request to the server, and subsequent requests go directly to the server.

Version Numbers. Application requirements tend to be dynamic and change over time. Sometimes the changes are simple, like the addition of new RPCs to the interface, and some-

times the changes are dramatic, like the removal of fields from an RPC call. In either case, the RPC mechanism must have a way of allowing the application developers to minimize the impact of changes to an interface on existing software.

Some RPCs allow for two types of changes, major and minor. A major change is when the new interface definition is not backward compatible with existing client software. This class of change is usually the redefinition of fields from an RPC definition. In this case, the existing client software would not work with the new service provider. Minor changes would include the addition of a new procedure call to an existing interface definition. Since the existing clients would never invoke the new RPC, its software would still be compatible with any server that provided the new service.

ONC+ RPC supports an allowable range of valid interface numbers. As such, it is possible to have interface number collisions at both compile time and runtime. To avoid collisions, SUN recommends that interfaces, or protocol specifications, be registered with Sun Microsystems (10).

A utility program generates the DCE RPC interface numbers at development time. Interface number collisions are eliminated because the utility program uses multiple varying input parameters to generate the number.

Platform Independence

Platform independence is important in distributed computing. Differences in internal data representation should not effect the results of remote operations. In other words, servers with one way of representing data must be able to process requests from clients with a completely different data representation. Without platform independence, distributed computing would work only with computers of identical data representation.

Host machines have several degrees of freedom in their internal representations of data (e.g., ASCII or EBCDIC character representation, check-sum bit representation, floating point representation, and byte ordering). For example, in byte ordering, the two popular data formats are big endian and little endian. Big endian computers store data in memory with the high-order byte in the lower address location, and little endian computers store data with the low-order memory location first. Therefore, some protocol is required for communication between computers of dissimilar memory storage.

There are at least two protocols that could be used to achieve platform independence (11):

- The first protocol defines a single standard format for all data communications. The sender converts the data from the current format to the standard format and sends the message to the receiver. The receiver, in turn, receives the information in standard format and converts it to its respective format for processing. In this protocol there are four conversions in each RPC: (1) for the client to send the data to the server; (2) for the server to convert the RPC data into the server's format; (3) for the server to convert any results into the standard format; and (4) for the client to convert the results into its format for processing.
- The second protocol, known as "receiver makes right", supports multiple standard data formats and lets the sender choose any one of the allowed formats to use when sending the message. In this scenario, the sender can use its data format, as long as it is one of the supported formats, for the transmission. Part of the protocol tells the receiver the translation scheme used in formatting the message, and the appropriate filter routine is called to reformat the message correctly.

In the first case there are always two data conversions per transmission, once before transmission and again after reception. In the "receiver makes right" case, the number of conversions per transmission can range from zero to two, depending on the internal data representations of both the originating and receiving hosts.

ONC+ RPC interfaces uses the eXternal Data Reference (XDR) protocol, which supports a single fixed-format data representation (single canonical format). DCE RPC interfaces use the "receiver makes right" multicanonical protocol (12).

Supporting Services

Supporting services provide common functionality that enables software designers and developers to create secure applications with a high degree of location transparency. Secure services means that developers do not have to develop custom security mechanisms. Location transparency means that there is no impact on the client configuration when the servers are moved from host to host.

Security. Distributed applications require security services to restrict access to shared resources to valid users and systems. The security service must provide the capabilities to identify users and external systems in a reliable fashion. The application logic must use the service and determine if the requester has the authority to perform the specified operation.

An additional requirement for the security service is to provide message encryption and decryption. These services are necessary because the RPC transmission may take place on an unsecured network, where it can be intercepted by an unintended recipient. The application must determine when message encryption is necessary and use the appropriate level of encryption to ensure optimal security.

Name Services. Distributed applications use name services to facilitate location of available servers. Servers register their names and locations with the name service. Clients make inquiries into the name service by server name and receive information to allow direct contact to the server. Use of the name service dictates the use of well-known server names between the clients and servers. Use of a name service allows clients to contact servers regardless of where the server is running. This is also known as location transparency.

Time Services. For a variety of reasons, clocks on all hosts participating in a distributed computing environment must be synchronized. Debugging distributed applications is easier when all cooperating hosts have nearly the same time. Also, many security packages, such are MIT's Kerberos, require very tight tolerances in clock differences.

ONC+ and DCE both support the Network Time Protocol, NTP, for synchronizing host clocks (13).

RPC Call Semantics

Call semantics define the behavior of the RPC from the client's perspective. Local procedure calls are invoked only once. If anything goes wrong, e.g., the server process or host crashes, the process is terminated, and both the caller and callee are terminated together. Unlike local procedure calls, RPCs may be invoked once, more than once, or not at all, because two processes are involved. For example,

- Server and network failures could cause an RPC either to time out or wait forever.
- After an RPC has been successfully processed by a server, the server could crash before the server stub routines have completed the reply.
- The client could trap the RPC time-out and issue the same request again.

Thus, in a distributed application using RPCs, the definition of proper RPC call semantics is important to provide for correct handling of potential error situations, particularly server crashes.

Idempotent RPCs can be executed many times without affecting application correctness. Idempotent services are typically read-only operations, and examples include RPCs that return the current time, RPCs that return the balance of a checking fund, and servers that return stock quotes.

Non-idempotent services usually read and write operations that could affect application correctness when executed more that once. Examples of this class of servers are bank account withdrawals, file servers, and application database servers.

There are at least three classes of RPC call semantics (14):

1. Exactly once, which means that the remote operation is executed once and only once. This class of service is difficult to achieve because of the effects of remoteness discussed earlier.
2. At most once, which means that the remote operation was executed once or it was not executed at all. To the client, this means that if the RPC returns successfully, the remote operation completed as expected. However, if the RPC times out and returns an error code, the client does not know if the remote operation was executed once or perhaps not at all. Non-idempotent servers have this RPC semantic.
3. At least once, which means the remote procedure was executed at least once, but maybe multiple times. With this semantic, clients can continuously send RPCs until a reply is received. The application correctness is not affected if the server processes one or more of the requests. Idempotent servers have this RPC semantic.

Call semantics can be defined both at the system and the application level.

Each server designer and developer must consider these semantics when developing applications. Given that there can be multiple RPC interfaces per interface definition, there may be a hybrid model in which some RPCs have at most once semantics and other RPCs have at least once semantics. For example, a bank account interface could have withdraw and deposit RPCs that are have at most once semantics and a get balance RPC that has at least once semantics.

Applications can be designed to make non-idempotent operations idempotent by building an application-specific protocol that prevents multiple processing of the same request. For example, each client and server interaction can be uniquely labeled. The server records all labels and verifies that all new labels have not been used before processing new RPC calls. While many such protocols can be designed and built, they add complexity to the application.

Applications that require true atomic, consistency, isolation, and durability (ACID) transaction semantics can obtain transaction services from various vendors. The transaction services shield the applications developer from the RPC call semantics while managing data transformation from state to state. Thus, the transaction semantics are not managed at the application level. However, the application could experience a penalty in performance due to the overhead of the transaction management software.

Neither ONC+ nor DCE directly support transaction semantics. However, they can be used to build transaction services.

Concurrency and Threads

One drawback of synchronous RPCs is that they are blocking calls. Distributed applications use multithreading to address the blocking issues. Multithreading allows the programmer to create multiple concurrent flows of control within an address space and perform multiple operations in a concurrent paradigm (15). When one thread of a process is waiting on an RPC reply (or any I/O), other threads within the process can be doing useful tasks, thus increasing the throughput of an application. Furthermore, the use of threads enables low system overhead, because using multiple threads within a process reduces the number and cost of context switches that the operating system performs on the process, compared with a multiprocess/shared-memory solution. Context switching between two processes is a lot more expensive, in terms of host resources, than context switching between multiple threads within a process. Concurrency can be realized without using threads, but to do so requires a more complex, high overhead, multiple process/shared memory processing model. Thus, a threads/RPC combination affords the best of both worlds—ease of use, and performance.

By using threads in the client/server model, server applications can service multiple clients concurrently. A client can use threads to make multiple simultaneous requests to a server or multiple servers. Each thread progresses independently using its resources (stack space and registers), periodically synchronizing with other threads and sharing the process-level resources (heap data) as necessary. Some threads continue processing while other threads wait for services, such as disk I/O or network packet reception.

Event Propagation

In the context of distributed computing, events are things that happen within the context of normal processing. Events may occur in all processes, are asynchronous in nature, and are optionally caught by predetermined event-handling routines. Processes that do not trap certain events terminate on reception of the event.

Remote computing introduces a new class of events and the concept of event propagation. The new class of events re-

sults from the effects of remoteness and therefore includes server and network failure. In event propagation, remote events are trapped and forwarded to the client for processing. Once the client receives the event, it can process the event information, often simply the type of event, to maximize application accuracy. Clients that do not trap events usually terminate abnormally.

Some programming languages implement certain events as exceptions. This leads to exception processing, a form of programming that allows blocks of operations to perform without having to check for exceptions, such as failures, on each statement (16). Rather, the programming language intercepts the exception, when generated, and invokes a predefined set of code to process the exception.

ONC+ does not support exception processing for remote events. This means that an application should check for remote errors on every remote operation. DCE RPC supports both individual RPC call error checking and exception processing on remote events.

SUMMARY

The RPC is a powerful technology used in interprocess communications. It simplifies the development of distributed applications. RPC allows a programmer to code the calling program the same way, regardless of whether the called program is running on the same host or on a remote host. RPC implementations provide platform independence, location transparency, security, and transport independence and run on a wide variety of commercial computer platforms.

BIBLIOGRAPHY

1. W. Richard Stevens, *UNIX Network Programming,* Englewood Cliffs, NJ: Prentice-Hall, 1990, pp. 693–695.
2. John Bloomer, *Power Programming with RPC,* Sebastopol, CA: O'Reilly & Associates, 1992, pp. 3–5.
3. Harold W. Lockhart, Jr., *OSF DCE Guide to Developing Distributed Applications,* New York: McGraw-Hill, 1994, pp. 47–48.
4. John Bloomer, *Power Programming with RPC,* Sebastopol, CA: O'Reilly & Associates, 1992, p. 58.
5. *ONC+ Developers Guide,* Mountain View, CA: SunSoft, 1995, p. 251.
6. *ONC+ Developers Guide, Rev 1.0,* Englewood Cliffs, NJ; Prentice-Hall, 1993.
7. D. E. Ruddock and B. Dasarathy, *Multithreading Programs: Guidelines for DCE Applications,* Piscataway, NJ: IEEE Software, January 1996, pp. 80–90.
8. John Bloomer, *Power Programming with RPC,* Sebastopol, CA: O'Reilly & Associates, 1992, p. 236.
9. *OSF DCE Application Development Guide, Rev 1.0,* Englewood Cliffs, NJ: Prentice-Hall, 1993.
10. John Bloomer, *Power Programing with RPC,* Sebastopol, CA: O'Reilly & Associates, 1992, p. 44.
11. W. Richard Stevens, *UNIX Network Programming,* Englewood Cliffs, NJ: Prentice-Hall, 1990, pp. 178–179.
12. John Bloomer, *Power Programming with RPC,* Sebastopol, CA: O'Reilly & Associates, 1992, p. 22.
13. John Bloomer, *Power Programming with RPC,* Sebastopol, CA: O'Reilly & Associates, 1992, p. 28.
14. W. Richard Stevens, *UNIX Network Programming,* Englewood Cliffs, NJ: Prentice-Hall, 1990, pp. 696–697.
15. D. E. Ruddock and B. Dasarathy, *Multithreading Prorams: Guidelines for DCE Applications,* Piscataway, NJ: IEEE Software, 1996, pp. 80–90.
16. *OSF DCE Application Development Guide, Rev 1.0,* Englewood Cliffs, NJ: Prentice-Hall, 1993, pp. 7-1 to 7-12.

DAVID E. RUDDOCK
RICHARD WIKOFF
RICHARD SAK

REMOTE SENSING. See ELECTROMAGNETIC SUBSURFACE REMOTE SENSING; INFORMATION PROCESSING FOR REMOTE SENSING; MICROWAVE REMOTE SENSING THEORY; OCEANIC REMOTE SENSING; VISIBLE AND INFRARED REMOTE SENSING.