Lectures on distributed systems

Remote Procedure Calls

Paul Krzyzanowski

Introduction, or what's wrong with sockets?

SOCKETS are a fundamental part of client-server networking. They provide a relatively easy mechanism for a program to establish a connection to another program, either on a remote or local machine and send messages back and forth (we can even use read and write system calls). This interface, however, forces us to design our distributed applications using a read/write (input/output) interface which is *not* how we generally design non-distributed applications. In designing centralized applications, the procedure call is usually the standard interface model. If we want to make distributed computing look like centralized computing, input-output-based communications is not the way to accomplish this.

In 1984, Birrell and Nelson devised a mechanism to allow programs to call procedures on other machines. A process on machine *A* can call a procedure on machine *B*. The process on *A* is suspended and execution continues on *B*. When *B* returns, the return value is passed to *A* and *A* continues execution. This mechanism is called the **Remote Procedure Call** (**RPC**). To the programmer, the goal is that it should appear as if a normal procedure call is taking place. Obviously, a remote procedure call is different from a local one in the underlying implementation.

Steps in a remote procedure call

Clearly, there is no architectural support for making remote procedure calls. A local procedure call generally involves placing the calling parameters on the stack and executing some form of a *call* instruction to the address of the procedure. The procedure can read the parameters from the stack, do its work, place the return value in a register and then return to the address on top of the stack. None of this exists for calling remote procedures. We'll have to simulate it all with the tools that we do have, namely local procedure calls and sockets for network communication. This simulation makes remote procedure calls a *language-level* construct as opposed to sockets, which are an *operating system level construct*. This means that our compiler will have to know that remote procedure call invocations need the presence of special code.

The entire trick in making remote procedure calls work is in the creation of *stub functions* that make it appear to the user that the call is really local. A stub function looks like the function that the user intends to call but really contains code for sending and receiving messages over a network. The following sequence of operations takes place (from p. 693 of W. Richard Steven's *UNIX Network Programming*):

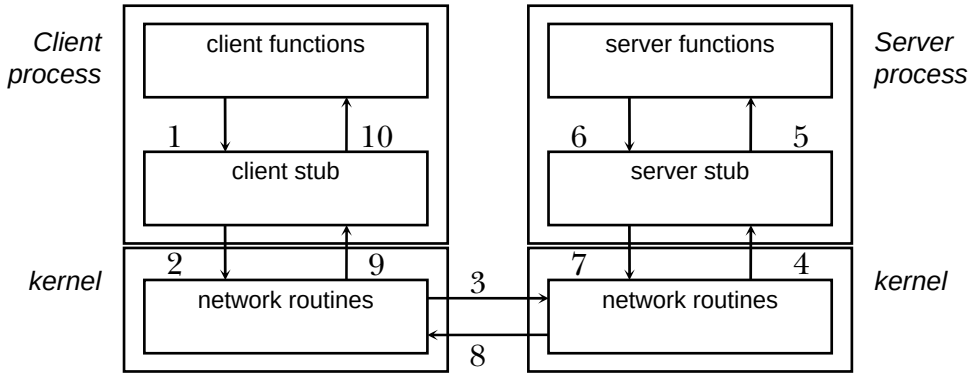

Figure 1. Functional steps in a remote procedure call

The sequence of operations, depicted in Figure 1, is:

1. The client calls a local procedure, called the *client stub*. To the client process, it appears that this is the actual procedure. The client stub packages the arguments to the remote procedure (this may involve converting them to a standard format) and builds one or more network messages. The packaging of arguments into a network message is called *marshaling*.
2. Network messages are sent by the client stub to the remote system (via a system call to the local kernel).
3. Network messages are transferred by the kernel to the remote system via some protocol (either connectionless or connection-oriented).
4. A *server stub* procedure on the server receives the messages. It *unmarshals* the arguments from the messages and possibly converts them from a standard form into a machine-specific form.
5. The server stub executes a local procedure call to the actual server function, passing it the arguments that it received from the client.
6. When the server is finished, it returns to the server stub with its return values.
7. The server stub converts the return values (if necessary) and marshals them into one or more network messages to send to the client stub.
8. Messages get sent back across the network to the client stub.

9. The client stub reads the messages from the local kernel.
10. It then returns the results to the client function (possibly converting them first).

The client code then continues its execution…

The major benefits of RPC are twofold: the programmer can now use procedure call semantics and writing distributed applications is simplified because RPC hides all of the network code into stub functions. Application programs don't have to worry about details (such as sockets, port numbers, byte ordering). Using the OSI reference model, RPC is a presentation layer service.

Several issues arise when we think about implementing such a facility:

How do you pass parameters?

Passing by value is simple (just copy the value into the network message). Passing by reference is hard – it makes no sense to pass an address to a remote machine. If you want to support passing by reference, you'll have to copy the items referenced to ship them over and then copy the new values back to the reference. If remote procedure calls are to support more complex structures, such as trees and linked lists, they will have to copy the structure into a pointerless representation (e.g., a flattened tree), transmit it, and reconstruct the data structure on the remote side.

How do we represent data?

On a local system there are no data incompatibility problems—the data format is always the same. With RPC, a remote machine may have different byte ordering, different sizes of integers, and a different floating point representation. The problem was solved in the IP protocol suiteby forcing everyone to use big endian[1] byte ordering for all 16 and 32 bit fields in headers (hence the *htons* and *htonl* functions). For RPC, we need to select a "standard" encoding for all data types that can be passed as parameters if we are to communicate with heterogeneous systems. Sun's RPC, for example, uses XDR (eXternal Data Representation) for this process. Most data representation implementations use *implicit typing* (only the value is transmitted, not the type of the variable). The ISO data representation (ASN.1—Abstract Syntax Notation) uses *explicit typing*, where the type of each field is transmitted along with the value.

[1] Big endian storage stores the most significant byte(s) in low memory. Little endian storage stores the most significant byte(s) of a word in high memory. Machines such as Sun Sparcs and 680x0s use big endian storage. Machines such as Intel x86/Pentium, Vaxen, and PDP-11s use little endian.

What should we bind to?

We need to locate a remote host and the proper process (port or transport address) on that host. Two solutions can be used. One solution is to maintain a centralized database that can locate a host that provides a type of service (proposed by Birell and Nelson in 1984). A server sends a message to a central authority stating its willingness to accept certain remote procedure calls. Clients then contact this central authority when they need to locate a service. Another solution, less elegant but easier to administer, is to require the client to know which host it needs to contact. A server on that host maintains a database of locally provided services.

What transport protocol should be used?

Some implementations allow only one to be used (e.g. TCP). Most RPC implementations support several and allow the user to choose.

What happens when things go wrong?

There are more opportunities for errors now. A server can generate an error, there might be problems in the network, the server can crash, or the client can disappear while the server is running code for it. The transparency of remote procedure calls breaks here since local procedure calls have no concept of the failure of the procedure call. Because of this, programs using remote procedure calls have to be prepared to either test for the failure of a remote procedure call or catch an exception.

What are the semantics of calling remote procedures?

The semantics of calling a regular procedure are simple: a procedure is executed exactly once when we call it. With a remote procedure, the “exactly once” aspect is quite difficult to achieve. A remote procedure may be executed:

- 0 times if the server crashed or process died before running the server code.
- once if everything works fine.
- once or more if the server crashed after returning to the server stub but before sending the response. The client won’t get the return response and may decide to try again, thus executing the function more than once. If it doesn’t try again, the function is executed once.
- more than once if the client times out and retransmits. It’s possible that the original request may have been delayed. Both may get executed (or not).

If a function may be run any number of times without harm, it is *idempotent* (e.g., time of day, math functions, read static data). Otherwise, it is a *nonidempotent* function (e.g., append or modify a file).

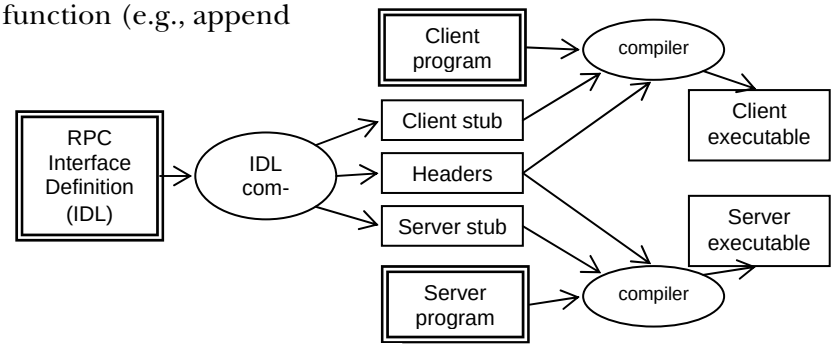

Figure 2. Compilation steps for Remote Procedure Calls

What about performance?

A regular procedure call is fast—typically only a few instruction cycles. What about a remote procedure call? Think of the extra steps involved. Just calling the client stub function and getting a return from it incurs the overhead of a procedure call. On top of that, we need to execute the code to marshal parameters, call the network routines in the OS (incurring a context switch), deal with network latency, have the server receive the message and switch to the server process, unmarshal parameters, call the server function, and do it all over again on the return trip. Without a doubt a remote procedure call will be much slower.

What about security?

This is something to worry about. More on this later…

Programming with remote procedure calls

Most popular languages today (C, C++, Java, Scheme, *et alia*) have no concept of remote procedures and are therefore incapable of generating the necessary stub functions. To enable the use of remote procedure calls with these languages, the commonly adopted solution is to provide a separate compiler that generates the client and server stub functions. This compiler takes its input from a programmer-specified definition of the remote procedure call interface. Such a definition is written in an *interface definition language*.

The interface definition generally looks similar to function prototype declarations: it enumerates the set of functions along with input and return parameters. After the RPC compiler is run, the client and server programs can be compiled and linked with the appropriate stub functions (Figure 2). The client procedure has to be modified to initialize the RPC mechanism (e.g. locate the server and possibly establish a connection) and to handle the failure of remote procedure calls.

Advantages of RPC

- You don't have to worry about getting a unique transport address (a socket on a machine). The server can bind to any port and register the port with its RPC name server. The client will contact this name server and request the port number that corresponds to the program it needs.
- The system is transport independent. This makes code more portable to environments that may have different transport providers in use. It also allows processes on a server to make themselves available over every transport provider on a system.
- Applications on the client only need to know one transport address—that of the *rpcbind* (or *portmap*) process.
- The function-call model can be used instead of the send/receive (read/write) interface provided by sockets.

The first generation of Remote Procedure Calls

Sun's RPC implementation (ONC RPC)

Sun Microsystem's was one of the first systems to provide RPC libraries and a compiler, developing it as part of their Open Network Computing (ONC) architecture in the early 1980's. Sun provides a compiler that takes the definition of a remote procedure interface and generates the client and server stub functions. This compiler is called ***rpcgen***. Before running this compiler, the programmer has to provide the interface definition. The interface definition contains the function declarations, grouped by version numbers (to support older clients connecting to a newer server) and a unique program number. The program number enables clients to identify the interface that they need. Other components provided by Sun are XDR, the format for encoding data across heterogeneous machines and a run-time library that implements the necessary protocols and socket routines to support RPC.

All the programmer has to write is a client procedure, the server functions, and the RPC specification. When the RPC specification (a file suffixed with `.x`, for example a file named `date.x`) is compiler with *rpcgen*, three or four files are created. These are (for `date.x`):

`date.h`	contains definitions of the program, version, and declarations of the functions. Both the client and server functions should include this file.
`date_svc.c`	C code to implement the server stub.
`date_clnt.c`	C code to implement the client stub.

`date_xdr.c` contains the XDR routines to convert data to XDR format. If this file is generated, it should be compiled and linked in with both the client and server functions.

If we look at the sample code (page 23), we notice that the first thing that must be done is to compile the data definition file `date.x` (page 24). After that, both the client and server functions may be compiled and linked in with the respective stub functions generated by *rpcgen*.

The client and server functions do not have to be modified much from a local implementation. The client must initialize the RPC interface with the function *clnt_create*, which creates an **RPC handle** to the specified program and version on a given host. It must be called before any remote procedure calls are issued. The parameters that we give it are the name of a server, the name of the program, the version of the program, and the transport protocol to use. Both the name and version of the program are obtained from the header file that was generated by *rpcgen*.

In older versions of, the transport protocol would be either the string "`tcp`" or the string "`udp`" to specify the respective IP service RPC (this is still supported and must be used with the Linux implementation of RPC). To make the interface more flexible, UNIX System V release 4 (SunOS ≥ 5) network selection routines allow a more general specification. They search a file (/etc/netconfig) for the first provider that meets your requirements. This last argument can be:

"`netpath`"	search a NETPATH environment variable for a sequence of preferred transport providers)
"`circuit_n`"	find the first virtual circuit provider in the NETPATH list), "`datagram_n`" (find the first datagram provider in the NETPATH list)
"`visible`"	find the first visible transport provider in /etc/netconfig)
"`circuit_v`"	find the first visible virtual circuit transport provider in /etc/netconfig)
"`datagram_v`"	find the first visible datagram transport provider in /etc/netconfig).

Each remote procedure call is restricted to accepting a single input parameter along with the RPC handle[2]. It returns a *pointer* to the result. The server functions have to be modified to accept a pointer to the value declared in the RPC definition (.x file) as an input and return a pointer to the result value. This pointer must be a pointer to static data (otherwise the area that is pointed to will be undefined when the procedure returns and the procedure's frame is freed). The names of the RPC procedures are the names in the RPC definition file converted to lower case and suffixed by an underscore followed by a version number. For example, BIN_DATE becomes a reference to the function *bin_date_1*. Your

[2] This restriction has now been removed and can be disabled with a command-line option to *rpcgen*.

server must implement *bin_date_1* and the client code should issue calls to *bin_date_1*.

What happens when we run the program?

The server

When we start the server, the server stub runs and puts the process in the background (don't forget to run *ps* to find it and kill it when you no longer need it)[3]. It creates a socket and binds any local port to the socket. It then calls a function in the RPC library, *svc_register*, to register the program number and version. This contacts the *port mapper*. The port mapper is a separate process that is usually started at system boot time. It keeps track of the port number, version number, and program number. On UNIX System V release 4, this process is *rpcbind*. On earlier systems, it was known as *portmap*.

The server then waits for a client request (i.e., it does a *listen*).

The client

When we start the client program, it first calls *clnt_create* with the name of the remote system, program number, version number, and protocol. It contacts the port mapper on the remote system to find the appropriate port on that system.

The client then calls the RPC stub function (*bin_date_1* in this example). This function sends a message (e.g., a datagram) to the server (using the port number found earlier) and waits for a response. For datagram service, it will retransmit the request a fixed number of times if the response is not received.

The message is then received by the remote system, which calls the server function (*bin_date_1*) and returns the return value back to the client stub. The client stub then returns to the client code that issued the call.

RPC in the Distributed Computing Environment (DCE RPC)

The Distributed Computing Environment (DCE) is a set of components designed by the Open Software Foundation (OFS) for providing support for distributed applications and a distributed environment. After merging with X/Open, this group is currently called The Open Group. The components include a distributed file service, a time service, a directory, service, and several others. Of interest to us here is the DCE remote procedure call. It is very similar to Sun's RPC. Interfaces are written in an interface definition language called *Interface Definition Notation (IDN)*. Like Sun's RPC, the interface definitions look like function prototypes.

[3] This is the default behavior. A command-line flag to *rpcgen* disables the automatic running as a daemon.

One deficiency in Sun's RPC is the identification of a server with a "unique" 32-bit number. While this is a far larger space than the 16-bit number space available under sockets, it's still not comforting to come up with a number and hope that it's unique. DCE's RPC addresses this deficiency by not having the programmer think up a number. The first step in writing an application is getting a unique ID with the *uuidgen* program. This program generates a prototype IDN file containing an interface ID that is guaranteed never to be used again. It is a 128-bit value that contains a location code and time of creation encoded in it. The user then edits the prototype file, filling in the remote procedure declarations.
After this step, an IDN compiler, similar to *rpcgen*, generates a header, client stub, and server stub.

Another deficiency in Sun's RPC is that the client must know the machine on which the server resides. It then can ask the RPC name server on that machine for the port number corresponding to the program number that it wishes to access. DCE supports the organization of several machines into administrative entities called *cells*. Every machine knows how to communicate with a machine responsible for maintaining information on cell services – the *cell directory server*.

With Sun's RPC a server only registers its {program number → port mapping} with a local name server (*rpcbind*). Under DCE, a server registers its endpoint (port) on the local machine with the RPC daemon (name server) as well as registering its {program name → machine} mapping with its cell directory server. When a client wishes to establish communication with an RPC server, it first asks its cell directory server to locate the machine on which the server resides. The client then talks to the RPC dæmon on that machine to get the port number of the server process. DCE also supports more complicated searches that span cells.

Second generation RPCs: object support

As object oriented languages began to gain popularity in the late 1980's, it was evident that both the Sun (ONC) and DCE RPC systems did not provide any support for instantiating remote objects from remote classes, keeping track of instances of objects, or providing support for polymorphism[4]. RPC mechanism still functioned but they did not support object oriented programming techniques in an automated, transparent manner.

Microsoft DCOM

In April 1992, Microsoft released Windows 3.1 which included a mechanism called OLE (object linking and embedding). This allowed a program to dynami-

[4] Polymorphism is the ability to create different functions with the same name. The appropriate function is invoked based on its parameters.

cally link other libraries to allow facilities such as embedding a spreadsheet into a Word document (this wasn't a Microsoft innovation – they were just trying to catch up to Apple). OLE evolved into something called COM (Component Object Model). A COM object is a binary file. Programs that use COM services have access to a standardized interface for the COM object (but not its internal structures). COM objects are named with globally unique identifiers (GUIDs) and classes of objects are identified with class IDs. Several methods exist to create a COM object (e.g., *CoGetInstanceFromFile*). The COM libraries look up the appropriate binary code (a DLL or executable) in the system registry, create the object, and return an interface pointer to the caller.

DCOM (Distributed COM) was introduced with Windows NT 4.0 in 1996 and is an extension of the Component Object Model to allow objects to communicate between machines. Since DCOM is meant to support access to remote COM objects, a process that needs to create an object would need to supply the network name of the server as well as the class ID. Microsoft provides a couple of mechanisms for accomplishing this. The most transparent is to have the remote machine's name fixed in the registry (or DCOM class store), associated with the particular class ID. This way, the application is unaware that it is accessing a remote object and can use the same interface pointer that it would for a local COM object. Alternatively, an application may specify a machine name as a parameter.

A DCOM server is capable of serving objects at runtime. A service known as the Service Control Manager (SCM), part of the DCOM library, is responsible for connecting to the server-side SCM and requesting the creating of the object on the server. On the server, a *surrogate process* is responsible for loading components and running them. This differs from RPC models such as ONC and DCE in that a service for a specific interface was not started a priori. This surrogate process is capable of handling multiple clients simultaneously.

To support the identification of specific instances of a class (individual objects), DCOM provides an object naming capability called a *moniker*. Each instance of an object can create its own moniker and pass it back to the client. The client will then be able to refer to it later or pass the moniker to other processes. A moniker itself is an object. Its *IMoniker* interface can be used to locate, activate, and access the bound object without having any information about where the object is located.

Several types of monikers are supported:

- File moniker: This moniker uses the file type (e.g., ".doc") to determine the appropriate object (e.g., Microsoft Word). Microsoft provides support for persistence - storing an object's data in a file. If a file represents a stored object, DCOM will use the class ID in the file to identify the object.
- URL moniker: This abstracts access to URLs via Internet protocols (e.g. *http, https, ftp*) in a COM interface. Binding a URL moniker allows the

remote data to be retrieved. Internally, the URL moniker uses the WinInet API to retrieve data.

- Class moniker: This is used together with other monikers to override the class ID lookup mechanism.

DCOM also provides some support for persistent objects via *monikers*.

Beneath DCOM: ORPC

Microsoft DCOM on its own does not provide remote procedure call capabilities. It is a set of libraries that assumes that RPC is available to the system. Beneath DCOM is Microsoft's RPC mechanism, called Object RPC (ORPC). This is a slight extension of the DCE RPC protocol with additions to support an *Interface Pointer Identifier* (IPID), versioning information, and extensibility information. The Interface Pointer Identifier is used to identify a specific instance of a class where the call will be processed. It also provides the capability for *referrals* – remote object references (IPIDs) can be passed around.

The marshaling mechanism is the same Network Data Representation (NDR) as in DCE RPC with one new type representing a marshaled interface (IPID). To do this marshaling, DCOM (and ORPC) needs to know methods, parameters, and data structures. This is obtained from an interface definition language (called *MIDL*, for *Microsoft Interface Definition Language*). As might be expected, this is identical to DCE's IDN with extensions for defining objects. The MIDL files are compiled with an IDL compiler that generates C++ code for marshaling and unmarshaling. The client side is called the proxy and the server side is called the stub. Both are COM objects that can be loaded by the COM libraries as needed.

Remote reference counting

A major difference for a server between object oriented programming and function-based programming is that functions persist while objects get instantiated and deleted from their classes (the code base remains fixed but data regions are allocated each time an object is created). For a server, this means that it must be prepared to create new objects and know when to free up the memory (destroy the objects) when the objects are no longer needed (there are no more references left to the object).

Microsoft DCOM does this explicitly rather than automatically. Object lifetime is controlled by *remote reference counting*. A call is made to *RemAddRef* when another reference to an object is added and *RemRelease* is called when a references is removed. The object itself is elided on the server when the reference count reaches zero.

This mechanism works fine but is not foolproof. If a client terminates abnormally, no messages were sent to decrement the reference count on objects

that the client was using. To handle this case, the server associates an expiration time for each object and relies on periodic messages from the client to keep the object reference alive. This mechanism is called *pinging*. The server maintains a ping frequency (pingPeriod) and a timeout period (numPingsToTimeOut). The client runs a background process that sends a ping set – the IDs of all remote objects on a specific server. If the timeout period expires with no pings received, all references are cleared.

DCOM summary

Microsoft DCOM is a significant improvement over earlier RPC systems. The Object RPC layer is an incremental improvement over DCE RPC and allows for object references. The DCOM layer builds on top of COM's access to objects (via function tables) and provides transparency in accessing remote objects. Remote reference management is still somewhat problematic in that it has to be done explicitly but at least there is a mechanism for supporting this. The moniker mechanism provides a COM interface to support naming objects, whether they are remote references, stored objects in the file system, or URLs. The biggest downside is that DCOM is a Microsoft-only solution. It also doesn't work well across firewalls (a problem with most RPC systems) since the firewalls must allow traffic to flow between certain ports used by ORPC and DCOM.

CORBA

Even with DCE fixing some of the shortcomings in Sun's RPC, certain deficiencies still remain. For example, if a server is not running, a client cannot connect to it to call a remote procedure. It is an administrator's responsibility to ensure that the needed servers are started before any clients attempt to connect to them. If a new service or interface is added to the system, there is no means by which a client can discover this. In some environments, it might helpful for a client to be able to find out about services and their interfaces at run-time. Finally, object oriented languages expect polymorphism in function calls (the function may behave differently for different types of data). Traditional RPC has no support for this.

CORBA (Common Object Request Broker Architecture) was created to address these, and other, issues. It is an architecture created by an industry consortium of over 500 companies called the Object Management Group (OMG). The specification for this architecture has been evolving since 1989. The goal is to provide support for distributed heterogeneous object-oriented applications. Objects may be hosted across a network of computers (a single object is not distributed). The specification is independent of any programming language, operating system, or network to enable interoperability across these platforms.

Under CORBA, when a client wishes to invoke an operation (method) on an object, it makes a request and gets a response. Both the request and response pass through the *object request broker* (ORB). The ORB represents the entire set of interface libraries, stub functions, and servers that hide the mechanisms for communication, activation, and storage of server objects from the client. It lets objects discover each other at run time and invoke services.

When a client makes a request, the ORB:

- marshals arguments (at the client).
- locates a server for the object. If necessary, it creates a process on the server end to handle the request.
- if the server is remote, transmits the request (using RPC or sockets).
- unmarshals arguments into server format (at the server).
- marshals return value (at the server)
- transmits the return if the server is remote.

unmarshals results at client.

An Interface Definition Language (IDL) is used to specify the names of classes, their attributes, and their methods. An IDL compiler generates code to deal with the marshaling, unmarshaling, and ORB/network interactions. It generates client and server stubs. A sample IDL is shown in Figure 3.

```
Module StudentObject {
    Struct StudentInfo {
        String name;
        int id;
        float gpa;
    };
    exception Unknown {};
    interface Student {
        StudentInfo getinfo(in string name)
            raises(unknown);
        void putinfo(in StudentInfo data);
    };
};
```

Figure 3. CORBA IDL example

Programming is most commonly accomplished via object reference and requests: clients issue a request on a CORBA object using the *object reference* and invoking the desired methods within it. For example, using the IDL in Figure 3 one might have code such as:

```
Student st = ... // get object reference
try {
    StudentInfo sinfo = st.getinfo("Fred Grampp");
} catch (Throwable e) {
     ... // error
}
```

Beneath the scenes, this code results in a stub function being called, parameters marshaled, and sent to the server. The client and server stubs can be used only if the name of the class and method is known at compile time. Otherwise, CORBA supports *dynamic binding* – assembling a method invocation at run time via the Dynamic Invocation Interface (DII). This interface provides calls to set the class, build the argument list, and make the call. The server counterpart (for

creating a server interface dynamically) is called the *Dynamic Skeleton Interface (DSI)*[5]. A client can discover names of classes and methods at run time via the *interface repository*. This is a name server that can be queried to discover what classes a server supports and which objects are instantiated.

CORBA standardized the functional interfaces and capabilities but left the actual implementation and data representation formats to individual ORB vendors. This led to the situation where one CORBA implementation might not necessarily be able to communicate with another. Applications generally needed some reworking to move from one vendor's CORBA product to another.

In 1996, CORBA 2.0 added interoperability as a goal in the specification. The standard defined a network protocol called *IIOP* (the *Internet Inter-ORB Protocol*) which would work across any TCP/IP based CORBA implementations. In fact, since there was finally a standardized, documented protocol, IIOP itself could be used in systems that do not even provide a CORBA API. For example, it could be used as a transport for an implementation of Java RMI (RMI over IIOP; RMI will be covered further on).

The hope in providing a well-documented network protocol such as IIOP along with the full-featured set of capabilities of CORBA was to usher in a wide spectrum of diverse Internet services. Organizations can host CORBA-aware services. Clients throughout the Internet will be able to query these services, find out their interfaces dynamically, and invoke functions. The pervasiveness of these services could be as ubiquitous as HTML web access.

CORBA summary

Basically, CORBA builds on top of earlier RPC systems and offers the following additional capabilities:

- Static or dynamic method invocations (RPC only supports static binding).
- Every ORB supports run time metadata for describing every server interface known to the system.
- An ORB can broker calls within a single process, multiple processes on the same machine, or distributed processes.
- Polymorphic messaging – an ORB invokes a function on a target object. The same function may have different effects depending on the type of the object.
- Automatically instantiate objects that are not running
- Communicate with other ORBs.

CORBA also provides a comprehensive set of services (known as COS, for *COrba Services)* for managing objects:

[5] Some systems use the term *stub* to refer to the client stub and *skeleton* to refer to the server stub. CORBA is one of them. Microsoft uses the term *proxy* for the client stub and *stub* for the server stub.

- Life-Cycle Services: provides operations for creating, copying, moving, and deleting components.
- Persistence Service (externalization): provides an interface for storing components on storage servers.
- Naming Service: allows components to locate other components by name.
- Event Service: components can register/unregister their interest in specific events.
- Concurrency Control Services: allows objects to obtain locks on behalf of transactions.
- Transaction Service: provides two-phase commit coordination (more on this later).
- Query Service: allows query operations on objects.
- Licensing Service: allows metering the use of components.
- Properties Service: allows names/properties to be associated with a component.

The price for the capabilities and flexibility is complexity. While CORBA is reliable and provides comprehensive support for managing distributed services, deploying and using CORBA generally has rather steep learning curve. Integrating it with languages is not always straightforward. Unless one could really take advantage of CORBA's capabilities, it is often easier to use a simpler and less powerful system to invoke remote procedures. CORBA enjoys a decent level of success, but only in pools of users rather than an Internet-wide community. CORBA suffered in being late to standardize on TCP/IP-based protocols and deploying Internet-based services.

Java RMI

CORBA aims at providing a comprehensive set of services for managing objects in a heterogeneous environment (different languages, operating systems, networks). Java, in its initial inception, supported the downloading of code from a remote site but its only support for distributed communication was via sockets. In 1995, Sun (the creator of Java) began creating an extension to Java called Java RMI (Remote Method Invocation). Java RMI enables a programmer to create distributed applications where methods of remote objects can be invoked from other Java virtual machines.

A remote call can be made once the application (client) has a reference to the remote object. This is done by looking up the remote object in the naming

service (the *RMI registry*) provided by RMI and receiving a reference as a return value. Java RMI is conceptually similar to RPC butt supports the semantics of object invocation in different address spaces.

One area in which the design of Java differs from CORBA and most RPC systems is that RMI is built for Java only. Sun RPC, DCE RPC, Microsoft's DCOM and ORPC, and CORBA are designed to be language, architecture, and (except for Microsoft) operating system independent. While those capabilities are lost, the gain is that RMI fits cleanly into the language and has no need for standardized data representations (Java uses the same byte ordering everywhere). The design goals for Java RMI are:

- it should fit the language, be integrated into the language, and be simple to use
- support seamless remote invocation of objects
- support callbacks from servers to applets
- preserve safety of the Java object environment
- support distributed garbage collection
- support multiple transports

The distributed object model is similar to the local Java object model in the following ways:

1. A reference to an object can be passed as an argument or returned as a result.
2. A remote object can be cast to any of the set of remote interfaces supported by the implementation using the Java syntax for casting.
3. The built-in Java `instanceof` operator can be used to test the remote interfaces supported by a remote object.

The object model differs from the local Java object model in the following ways:

1. Classes of remote objects interact with remote interfaces, never with the implementation class of those interfaces.
2. Non-remote arguments to (and results from) a remote method invocation are passed by copy, not by reference.
3. A remote object is passed by reference, not by copying the actual remote implementation.
4. Clients must deal with additional exceptions.

Interfaces and classes

All remote interfaces extend the interface `java.rmi.Remote`. For example:

```
public interface bankaccount extends Remote
{
     public void deposit(float amount)
            throws java.rmi.RemoteException;
```

```
        public void withdraw(float amount)
              throws OverdrawnException,
              java.rmi.RemoteException;
}
```

Note that each method must declare `java.rmi.RemoteException` in its `throws` clause. This exception is thrown whenever the remote method invocation fails.

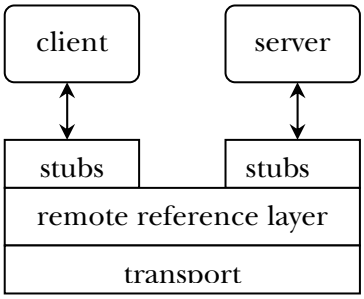

Figure 4. Java RMI architecture

Remote Object Class

The `java.rmi.server.RemoteObject` class provides remote semantics of `Object` by implementing the `hashCode`, `equals`, and `toString`[6]. The functions needed to create objects and make them available remotely are provided by `java.rmi.server.RemoteServer` and subclasses. The `java.rmi.server.UnicastRemoteObject` class defines a unicast (single) remote object whose references are valid only while the server process is alive.

Stubs

Java RMI works by creating stub functions. The stubs are generated with the `rmic` compiler.

Locating objects

A bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based methods of the class `java.rmi.Naming`. For example,

```
BankAccount acct = new BankAcctImpl();
String url = "rmi://java.sun.com/account";
// bind url to remote object
java.rmi.Naming.bind(url, acct);

// look up account
acct = (BankAccount)java.rmi.Naming.lookup(url);
```

RMI architecture

RMI is a three-layer architecture (Figure 4). The top layer is the *stub/skeleton layer.* It transmits data to the remote reference layer via *marshal streams.* Marshal streams employ a method called object serialization, which enables objects to be transmitted between address spaces (passing them by copy, unless they are remote objects which are passed by reference).

The client stub performs the following steps:

[6] the `toString` method returns the reference of the object as a string.

1. initiates a call to remote object
2. marshals arguments
3. informs remote reference layer that the call should be invoked
4. unmarshals return value or exception
5. informs remote reference layer that the call is complete.

The server stub (skeleton):

1. unmarshals arguments
2. makes up-call to the actual remote object implementation
3. marshals the return value of the call (or exception)

The stub/skeleton classes are determined at run time and dynamically loaded as needed. They must be precompiled with the `rmic` compiler.

The *remote reference layer* deals with the lower-level transport interface. It is responsible for carrying out a specific remote reference protocol that is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own remote reference subclass. Various protocols are possible. For example:

- unicast point-to-point
- invocation to replicated object groups
- support for a specific replication strategy
- support for a persistent reference to a remote object (enabling activation of the remote object)
- reconnection strategies

Multicast delivery is not part of JDK 1.1 RMI.

The *transport layer* is the transport-specific part of the protocol stack. It:

- sets up connections, manages connections
- monitors connection liveness
- listens for incoming calls
- maintains a table of remote objects that reside in the address space
- sets up a connection for an incoming call
- locates the dispatched for the target of a remote call and passes the connection to the dispatcher.

Third generation RPCs: Web services and the XML bandwagon

XML RPC

SOAP

Microsoft .NET

Appendix A: Sun's RPC definition language

Program identification

Every program (a collection of RPC procedures) is identified by some value. This value is a 32-bit integer that you have to select. The restrictions imposed on this number are:

`0x00000000–0x1fffffff`	defined by Sun for standard services
`0x20000000–0x3fffffff`	defined by the user
`0x40000000–0x5fffffff`	transient processes
`0x60000000–0xffffffff`	reserved for future use

A program identification is:

```
program identifier {
      version_list
} = value;
```

The *identifier* names the program. The version list is a list of definitions, each of the form:

```
version identifier {
      procedure_list
} = value;
```

The *value* is an unsigned integer. Usually you only have one version in a program definition. The *identifier* is a string that names the version of the program. The *rpcgen* compiler will generate a **#define** for it and the program identifier in the header file so that these values can be passed as arguments to *clnt_create*.

Every version definition contains a list of procedure. This procedure list contains a sequence of definitions, each of the form:

```
data_type procedure_name ( data_type ) = value;
```

The ***procedure_name*** is a string naming the procedure. The *value* is an unsigned integer that specifies the procedure number (0 is reserved for the "null" procedure). The ***data_type*** can be any simple C data type (such as **int**, **unsigned int**, **void**, or **char**) or a complex data type.

Data types

Constants may be used in place of an integer value. Constant definitions are converted to a **#define** statement by *rpcgen*:

```
const MAXSIZE = 512;
```

Structures are like C structures. *rpcgen* transfers the structure definition and adds a **typedef** for the name of that structure. For example,

```
struct intpair { int a, b };
```

is translated into:

```
struct intpair { int a, b };
typedef struct intpair intpair;
```

Enumeration types are also similar to C:

```
enum state { BUSY=1, IDLE=2, TRANSIT=3 };
```

Unions are not like C. A union is a specification of data types based on some criteria:

```
union identifier switch ( declaration ) {
      case_list
}
```

For example:

```
const MAXBUF=30;
union time_results switch (int status) {
      case 0: char timeval[MAXBUF];
      case 1: void;
      case 2: int reason;
}
```

If you set **status** to 0, you must assign data to **time_results_u.timeval** .

Type definitions are like C, for example,

```
typedef long counter;
```

Arrays are similar to C but may be of a fixed or variable length. A declaration of

```
int proc_hits[100];
```

defines a fixed-size array of 100 integers. A declaration such as:

```
long x_vals<50>;
```

defines a variable size array with a maximum size of 50 longs. The number may be omitted if there is no bound on the size. This declaration is translated to:

```
typedef struct {
            u_int x_vals_len;
            long *x_vals_val;
} x_coords;
```

Pointers are like C. However, a pointer is not sent over the network (the value would be meaningless on the other machine). What is sent is a boolean value (true for pointer, false for null) followed by the data to which the pointer points.

Strings are declared as if they were variable length arrays:

```
string name<50>;
```

defines a string of at most 50 characters. When implementing strings, you have to allocate space to store the string (for example, point to a static buffer when returning a string value). The value in angle brackets may be empty to state that there is no maximum length for the string. A string declaration is translated into a pointer to a character by *rpcgen*:

```
char **name;
```

Boolean data is defined as:

```
bool busy;
```

The variable **busy** may be set to **TRUE** or **FALSE**.

Opaque data is untyped data that contains an arbitrary sequence of bytes. It may be of a fixed or variable length:

```
opaque extra_bytes[512];
opaque more<512>;
```

The second definition is translated by rpcgen into:

```
struct {
      uint more_len;    /* length of the array */
      char *more_val;   /* space used by the array */
}
```

Writing procedures using Sun's RPC

- Create a procedure whose name is the name of the RPC definition but in lowercase with an underscore and version number after it.
- The argument to the procedure is a pointer to the argument data type specified in the RPC definition language.
- The procedure must return a pointer to the data type specified in the RPC definition language.
- Lower-level RPC routines use the procedure's return value after the procedure ends, so the return address must be that of a static variable.
- If a procedure allocates memory (e.g., to hold a string or linked list), the memory must not be freed until all processing of the data is complete. Data is translated to XDR after the procedure ends so you should only free the space on the **next** invocation of the procedure (you may need to set a static variable to keep track of this state).

External Data Representation

RPC routines call the XDR routines to translate data into and out of XDR format. The user normally need not worry about XDR—it's completely transparent.

However, you must use the *xdr_free* routine to free data that the procedure allocates. *xdr_free* takes a pointer to an XDR function and a data pointer. It frees memory that is associated with the data pointer. The function passed as a parameter is the XDR routine corresponding to the data type. The RPC compiler names these functions by prefacing the name of the type with **xdr_**. For example, if you define a structure named **id**, the corresponding XDR function is **xdr_id()**. If you have a linked list structure:

```
struct list {int val; struct list *next; };
```

and **head** is a pointer to the start of the list (you allocated memory for it). After the XDR routines translate the data, you can **use xdr_free(xdr_list, head)** to free the data.

Sun RPC example

Here's a sample program from Richard Stevens' text. There are four components: **Makefile** (so that we can run the *make* command and compile everything), **date.x** (the RPC definition), **server.c** (the server function), and **client.c** (the client program). The program is a simple one: the client program (*client*) accepts a machine name as an argument. It is assumed that before *client* is run the server program (*server*) is running on that machine. The client first requests the time from the server, which is returned as a 32-bit value. It then sends that result back to the server to get an ASCII string containing the date/time represented by that value.

Makefile

```
all: client server

client: client.o date_clnt.o date.h
     cc -o client client.o date_clnt.o -lnsl

server: server.o date_svc.o date.h
     cc -o server server.o date_svc.o -lnsl

date_svc.o:
     $(CC) $(CFLAGS) -c date_svc.c

date_clnt.o:
     $(CC) $(CFLAGS) -c date_clnt.c

client.o: date.h

server.o: date.h

date.h: date.x
     rpcgen date.x

clean:
```

```
	rm -f client client.o server server.o date_clnt.*
date_svc.* date.h

tar:
	tar cvf rpcdemo.tar date.x client.c server.c Makefile
```

date.x

```
/* date.x - description of remote date service */

/* we define two procedures:                              */
/*   bin_date_1 returns the time in binary format          */
/*         (seconds since Jan 1, 1970 00:00:00 GMT)   */
/*   str_date_1 converts a binary time to a readable date string
*/

program DATE_PROG {
     version DATE_VERS {
          long BIN_DATE(void)  = 1;     /* procedure number = 1
*/
          string STR_DATE(long)  = 2;   /* procedure number = 2
*/
     } = 1;
} = 0x31415926;
```

server.c

```
#include <rpc/rpc.h>
#include "date.h"

/* bin_date_1 returns the system time in binary format */
long *
bin_date_1()
{
     static long timeval;    /* must be static!! This value is
*/
                        /* used by rpc after bin_date_1 returns
*/
     long time();            /* Unix time function; returns time
*/

     timeval = time((long *)0);
     return &timeval;
}

/* str_date_1 converts a binary time into a date string */
char **
str_date_1(bintime)
long *bintime;
{
     static char *ptr; /* return value... MUST be static! */
     char *ctime();    /* Unix library function that does the
work */
     ptr = ctime(bintime);
     return &ptr;
}
```

client.c

```
/* client code */

#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include "date.h"

main(argc, argv)
int argc;
char **argv;
{
     CLIENT *cl; /* rpc handle */
     char *server;
     long *lresult;    /* return from bin_date_1 */
     char **sresult;   /* return from str_date_1 */

     if (argc != 2) {
           fprintf(stderr, "usage:  %s hostname\n", argv[0]);
           exit(1);
```

```
        }
        server = argv[1]; /* get the name of the server */
        /* create the client handle */
        if ((cl=clnt_create(server, DATE_PROG,
                                DATE_VERS, "netpath")) == NULL) {
              /* failed! */
              clnt_pcreateerror(server);
              exit(1);
        }
        /* call the procedure bin_date */
        if ((lresult=bin_date_1(NULL, cl))==NULL) {
              /* failed ! */
              clnt_perror(cl, server);
              exit(1);
        }
        printf("time on %s is %ld\n", server, *lresult);
        /* have the server convert the result to a date string */
        if ((sresult=str_date_1(lresult, cl)) == NULL) {
              /* failed ! */
              clnt_perror(cl, server);
              exit(1);
        }
        printf("date is %s\n", *sresult);
        clnt_destroy(cl); /* get rid of the handle */
        exit(0);
}
```

References

The Component Object Model Specification, Draft version 0.9, October 24, 1995, © 1992-1995 Microsoft Corp, http://www.microsoft.com/oledev/olecom/title.htm [probably more than you'll want to know about Microsoft's COM]

CORBA Architecture, version 2.1, Object Management Group, August 1997, pp 1-1 – 2-18 .[introductory CORBA concepts straight from the horse's mouth]

The OSF Distributed Computing Environment: Building on International Standards – A White Paper, Open Software Foundation, April 1992. [introduction to DCE and DCE's RFS]

Networking Applications on UNIX System V Release 4, Michael Padovano, ©1993 Prentice Hall. [guide to sockets, Sun RPC, and Unix network programming]

UNIX Network Programming, W. Richard Stevens, ©1990 Prentice Hall. [guide to sockets, Sun RPC, and Unix network programming]

JAVA™ Remote Method Invocation (RMI), ©1995-1997 Sun Microsystems, http://java.sun.com/products/jdk/rmi/index.html [probably all you'll want to know about Java RMI]

Distributed Operating Systems, Andrew Tanenbaum, © 1995 Prentice Hall, pp. 68-98, 520-524,535-540. [introductory information on sockets and RPC]

Modern Operating Systems, Andrew Tanenbaum, ©1992 Prentice Hall, pp. 145- 180, 307-313, 340-346. [introductory information on sockets and RPC]