Operating Systems

Remote Procedure Calls

By Paul Krzyzanowski
December 1, 2010

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). Moreover, they are the only interface to the network that most operating systems provide.

The sockets interface, however, forces us to design our distributed applications using a read/write (input/output) interface which is not how we generally think about application design and how different funcitonal blocks of an application communicate. In design single-process applications, the procedure call is usually the standard, most popular, and most familiar interface model. If we want to make distributed computing look like centralized comput- ing, input-output-based streams are not the way to accomplish this.

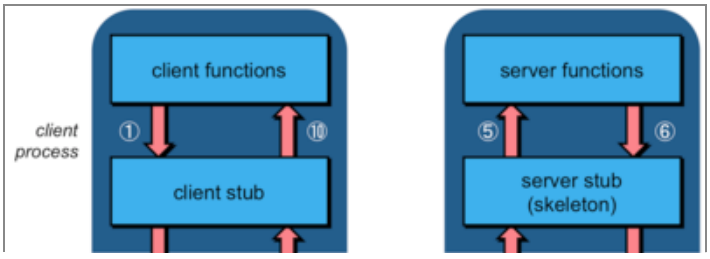
In 1984, Birrell and Nelson devised a mechanism to allow programs to call procedures on other machines. A process on machine *A* can call a procedure on machine *B*. The process on *A* is suspended and execution continues on *B* . When *B* returns, the return value is passed to *A* and *A* continues 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.

Implementing remote procedure calls

Let's think about how local procedure (function) calls work. 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 will then read the parameters from the stack, allocate more stack space for local variables, do its work, place the return value in a register, readjust the stack pointer, and then return to the address on top of the stack. The instructions to perform these operations are generated by the compiler and then executed by the processor.

Processors do not offer us architectural support for making remote calls of any sort. We'll have to simulate the entire process with the tools that we do have, namely local procedure calls and operating systems sockets for network communication. Operating systems do not give us a remote procedure call facility but rather access to the network via sockets. Just as the instructions for local procedure calls are generated by the compiler, the instructions for invoking remote procedures also have to be generated by the compiler. This 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 a local one. A stub function has the same interface as the remote function that the user intends to call but really contains just code for gathering parameters, sending, and receiving messages over a network. Figure 1 shows the sequence of operations that takes place in making a remote procedure call (adapted from p. 693 of W. Richard Steven's *UNIX Network Programming*). The sequence of operations is:

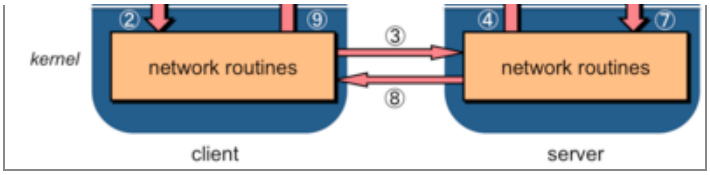


Figure 1. RPC flow

1. The client calls a local procedure, called the **client stub**. To the client process, it appears that this is the actual remote procedure, which it can call using a local procedure call (since the stub is a local 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 (by writing to a socket via system calls 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** process, sometimes called a **skeleton**, on the server receives the messages. It **unmarshals** the arguments from the messages and, if needed, converts them from a standard format into a machine-specific format.
5. The server stub calls a local procedure call to the actual server function, passing it the arguments that it received from the client. The server function gets the illustion that it was called locally by the client since the server stub calls it locally with the same parameters that the client transmitted.
6. When the server is finished, it returns to the server stub with its return values.
7. The server stub converts the return values to a standard format (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 feels that it just received a return value from the remote function, unaware that all the network messaging took place.

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. Also, application programs don't have to worry about details (such as sockets, port numbers, byte ordering). Using the OSI reference model, RPC can be viewed as 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. A memory location in a process on one machine is meaningless to another process on another machine. If we want to support passing by reference, we will have to copy the referenced items in order to ship them over in network messages 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 representa- tion (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 dealt with in the IP protocol suite by forcing everyone to use big endian byte ordering for all 16- and 32-bit fields in headers (hence the use of *htons* and *htonl* functions when you manipulate IP fields). 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. This is an implementation that uses **implicit typing**, where only the value is transmitted, not the name or type of the variable). Formats such as the ISO data representation format (ASN.1—Abstract Syntax Notation) and SOAP (Simple Object Access Protocol) use **explicit typing**, where the type of each field is transmitted along with the value.

> 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. The Intel architecture uses little endian. A number of architectures, such as ARM, allow you to set a flag in the processor to indicate which mode you want to use for data storage.

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 called an **idempotent** function. Examples are functions such as time of day, math functions, read static data). Otherwise, the function is a **nonidempotent function**. An example is a funciton that appends to or modifies a file.

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. The overhead of a remote procedure call will be much thousands of times slower than a regular procedure call.

Programming with remote procedure calls

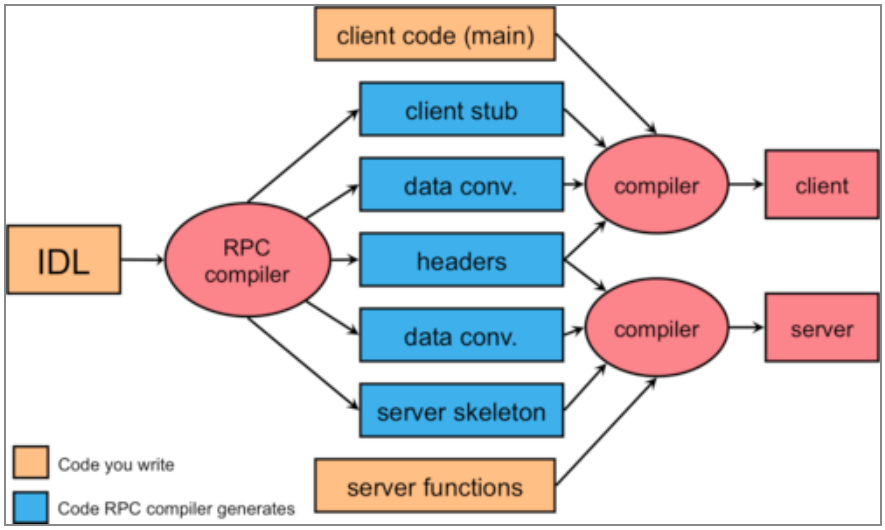


Figure 1. RPC complication

Although RPC is a language construct, it turns out that most popular programming languages (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** (**IDL**).

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 remote procedure calls

- 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 name server process. They don't need to know the port number of each server-side process that they want to contact.
- A function-call interface can be used instead of the send/receive (read/write) interface provided by sockets.
