A Large-Parameter Remote Procedure Call Mechanism in Unix

M. Satyanarayanan
Information Technology Center
Carnegie-Mellon University

The Problem

This document describes a ***Remote Procedure Call*** (RPC) mechanism for communications between workstations and a variety of central services. The term "RPC", as used here, connotes the following:[1]

- It is an abstraction, representing a style of communication between a ***Client*** process on one network node, and a ***Server*** process on another node. Each of these nodes may potentially support multiple clients and servers.

- There is no attempt to integrate RPC features into the programming language in use at the client and server sites. In particular, the languages in use at the two sites may be different. A network-wide standard is used to represent data types and the RPC mechanism is implemented by a runtime library.

- Interactions begin with a rendezvous between a client and a server. This is followed by an alternating sequence of RPC requests by the client, and replies by the server. Each request consists of an ***Operation*** and a set of ***Parameters***. A reply consists of a ***Return Code*** and a set of result parameters.

- Some of the transmitted RPC parameters may be very large in size. For example, when dealing a file server, entire files may be parameters. In the future, requests to other servers may involve parameters representing other kinds of large objects.

- The actual transfer of these large parameters may involve interactions with other subsystems at the client and server sites. For instance, transmission of a file is **NOT** viewed as a movement of a string of bytes between the address spaces of the client and server. Instead, it is viewed as the movement of that string of bytes between two file systems, mediated by the server and client processes. ⟶ RPC with file side effects

The performance and code structuring implications of the last two requirements have a strong influence on the proposed mechanism.

The Proposed Mechanism

A logical RPC connection between a server and a client is associated with two logical ***Channels***: a ***Request/Reply Channel*** and a ***Bulk-Transfer Channel***. Each of the channels is bidirectional, and uses protocols tailored to its function. The bulk-transfer channel is used for the sole purpose of transmitting large-sized parameters, generically referred to as ***Bulk-Units***. All other data associated with an RPC connection are transmitted over the request/reply channel.

---

[1] An in-depth discussion of RPC can be found in [2], while experiences with a specific implementation are described in [1].

ITALF

Neither server nor client code is aware of the fact that there are two logical channels. Each of them sees a single RPC connection, modelled by a set of interface calls supported by a ***Stub*** at each end. The initiation and coordination of activities on the two logical channels is handled entirely within the stubs. Conversely, the stubs are ignorant of the operations being requested by the client, and the parameters being passed between client and server.

A server can examine an RPC request before bulk-units are actually transferred. Consequently, the transfer of bulk-units is always initiated by the server. We assume that a given RPC connection deals with only one kind of bulk-unit.

A ***Bulk-Unit Descriptor*** is an RPC data type that is a placeholder for a bulk-unit in a parameter list. It contains the information needed by the stubs to define a complete execution instance of the corresponding bulk-transfer protocol. For example a file transfer bulk-descriptor will typically contain an opcode indicating whether a file is to be sent or received, the file name on the client side, the file name on the server side, and other optional information such as the encryption key to be used when transmitting the file. Bulk-unit descriptors are created and partially filled by the client, and sent in an RPC request to the server. The server examines the request, fills in the missing pieces of information in bulk-descriptors, and requests its stub to carry out the bulk-unit transfers. On completion of all bulk-transfers, a response is communicated back to the client.

The logical separation of bulk-transfers from other activities is motivated by the observation that the large size of bulk-units typically necessitates the use of a specialized protocol for their transmission. If a single logical channel were to be used, the stubs would have to be complex finite-state machines, representing nested protocols.

Using two logical channels breaks the stub into two simple pieces, one for each channel. In the absence of errors, the coordination of these channels is trivial. When an error does occur, more effort is needed to clean up state. Thus this approach streamlines and simplifies the normal case, concentrating the complexity in the area of error handling.

Implementing the Mechanism in Unix

In the 4.2BSD version of Unix, communication between processes on the same or different network nodes is supported via an abstraction called a ***Socket***. Sockets have unique network-wide ***Socket Addresses***, as well as process-specific ***Socket Descriptors***. The Unix kernel maintains the binding between descriptors and addresses, and multiplexes the physical network among all the sockets on a given network node.

A socket may be parameterized with respect to its ***Domain***, and its ***Type***. The former attribute determines the format of a socket address, while the latter specifies the service guarantees accorded to transmissions via that socket. Incorporating new socket domains or new socket types is a fairly major operation, involving extensions to the Unix kernel.

The stubs in the proposed RPC mechanism will be implemented as code above the Unix kernel. Each Unix process will have its own copy of stub code and data linked in.

A key implementation issue is the mapping of the logical request/reply and bulk-transfer channels into sockets. The obvious approach is taken here: two sockets are used, one for each logical channel. The types of sockets for each of the channels can be chosen to take advantage of the characteristics of data transmission on them. For example, requests and responses are brief and are unlikely to require sophisticated flow control. File transfer, on the other hand, may require extensive sequencing, reassembly and flow control.

It has been suggested that this use of two sockets to implement a single RPC connection is counterproductive for two reasons:

- One reason is a flavour of the "Two is harder to manage than one" argument, which basically says that keeping track of and coordinating two loci of communication is harder than multiplexing all communication on one locus.

- The second reason cited is that the use of more than the bare minimum of sockets is an unnecessary consumption of resources.

The decision to persist in using two sockets is based on the following observations:

- *Two is not always harder to deal with than one.* In particular, if the code dealing with each of the two logical channels is physically distinct, each is individually likely to be simpler than code that multiplexes a single socket. Since we are proposing a synchronous RPC mechanism, the amount of coordination needed between the two channels is minimal. As mentioned earlier, even if error handling turns out to be more complex, it is worth simplifying and optimizing the error-free path, which is the one used most often.

- *Multiplexing two logical channels on one socket involves the insertion and recognition of information disambiguating the two channels.* This complicates the format of the data being transmitted. Further, it is redundant because the Unix kernel is perfectly capable of performing the multiplexing and probably does it more efficiently than one can in a user-level process.

- *Using a single socket means that the path from the kernel interface to the actual network cannot be independently optimized for bulk- and non-bulk transmissions.* Any attempt to perform such an optimization would necessarily complicate the code within the kernel that implements sockets, because it would have to recognize and optimally deal with a

variety of cases.

- *By writing the code from the very beginning to use two sockets, it will be much easier to experiment with a variety of strategies.* If only one socket is used initially, it is inevitable that structural changes to stub code would be necessary to allow experimentation with multi-socket implementations later.

Two resources are impacted by the use of an extra socket per RPC connection: a socket descriptor, and memory for the data structure representing the socket. Neither of these is a sufficiently precious and scarce resource to warrant insistence on a single socket per RPC connection.

A Fringe Benefit

Maintaining the distinction between the two logical channels in the actual implementation yields an interesting spin-off: *bulk-units may be transferred to a site other than the client, in a manner that is transparent to the server.* The significance of this observation can be best appreciated in the context of diskless workstations. Suppose such a workstation *W* is on a local area network with a file server *F* on one node and a disk server *D* on another node. One could set up an RPC connection with the request/reply channel between *W* and *F*, and the bulk-transfer channel between *F* and *D*. Once initiated by a request from *W* to *F*, the transfer of a bulk-unit would proceed directly between *F* and *D*. The key observation here is that neither the server nor client code need be aware of this dichotomy of sites: the dependency is internal to the RPC stubs.

Whether this potential benefit is really worthwhile can best be determined by actual experimentation. Using two sockets to implement an RPC connection leaves open this option, if one chooses to exercise it. Multiplexing both channels on one socket gives up this option, *a priori.*

Summary

In conclusion, the major points made in this brief document are:

- The efficient transmission of large parameters, lazily-evaluated, is a critical consideration in the design of the proposed RPC mechanism.
- The use of two logical channels simplifies the structuring of the RPC stub code.
- Mapping each logical channel onto a separate Unix socket paves the way for valuable performance optimizations.
- The independent mapping also suggests a method for efficiently handling diskless workstations.
- The resources expended to achieve all these benefits is minimal. The key resource

expended is, in fact, conceptual and not material.

References

[1] Birrell, A.D. and Nelson, B.J.
Implementing Remote Procedure Calls Efficiently.
In *Preprints of the Ninth Symposium on Operating Systems Principles*. ACM, October, 1983.

[2] Nelson, B.J.
Remote Procedure Call.
Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981.