Palo Alto Research Center

Implementing Remote Procedure Calls

Andrew D. Birrell and Bruce Jay Nelson

XEROX

Implementing Remote Procedure Calls

Andrew D. Birrell and Bruce Jay Nelson

CSL-83-7 December 1983 [P83-00008]

Abstract: Remote procedure calls (*RPC*) appear to be a useful paradigm for providing communication across a network between programs written in a high level language. This paper describes a package providing a remote procedure call facility, the options that face a designer of such a package, and the decisions we made. We describe the overall structure of our RPC mechanism, our facilities for binding RPC clients, the transport level communication protocol, and some performance measurements. We include descriptions of some optimisations we used to achieve high performance and to minimize the load on server machines that have many clients.

A version of this paper will appear in *Transactions on Computer Systems*, vol. 2, no. 1, February, 1984.

CR Categories and Subject Descriptors: C.2.2 **[Computer Communication Networks]:** Network Protocols – *protocol architecture;* C.2.4 **[Computer Communication Networks]:** Distributed Systems - *distributed applications, network operating systems;* D.4.4 **[Operating Systems]:** Communications Management - *message sending, network communication;* D.4.7 **[Operating Systems]:** Organization and Design - *distributed systems.*

Keywords: Design, experimentation, performance, security, remote procedure calls, transport layer protocols, distributed naming and binding, inter-process communication, performance of communication protocols.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

Background

The idea of *remote procedure calls* (hereinafter called *RPC*) is quite simple. It is based on the observation that procedure calls are a well known and well understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across a communication network. When a remote procedure is invoked, the calling environment is suspended, the parameters are passed across the network to the environment where the procedure is to execute (which we will speak of as the *callee*), and the desired procedure is executed there. When the procedure finishes and produces its results, the results are passed back to the calling environment, where execution resumes as if returning from a simple single-machine call. While the calling environment is suspended, other processes on that machine may (possibly) still execute (depending on the details of the parallelism of that environment and the RPC implementation).

There are many attractions to this idea. One is the clean and simple semantics: these should make it easier to build distributed computations, and easier to get them right. Another is efficiency: procedure calls seem simple enough that the communication might be quite rapid. A third is the generality: in single machine computations, procedures are often the most important mechanism for communication between parts of the algorithm.

The idea of RPC has been around for many years. It has been discussed in the public literature many times since at least 1976 [15]. Nelson's doctoral thesis [13] is an extensive examination of the design possibilities for an RPC system and has references to much of the previous work on RPC. However, full-scale implementations of RPC have been far fewer than paper designs. Notable recent efforts include *Courier* in the Xerox NS family of protocols [4], and the current work at MIT [10].

This paper results from the construction of an RPC facility for the *Cedar* project. Before embarking on this, we felt that because of earlier work (particularly Nelson's thesis and associated experiments) we understood the choices that a designer of an RPC facility must make. Our task was to make the choices in light of our particular aims and environment. In practice, we found that several areas were inadequately understood, and we produced a system whose design has several novel aspects. Major issues facing the designer of an RPC facility include: the precise semantics of a call in the presence of machine and communication failures; the semantics of address-containing arguments in the (possible) absence of a shared address space; integration of remote calls into existing (or future) programming systems; binding (how a caller determines the location and identity of the callee); suitable protocols for transfer of data and control between caller and callee; and how to provide (if desired) data integrity and security in an open communication network. In building our RPC package we addressed each of these issues, but it is not possible to describe all of them in suitable depth in a single paper. This paper includes a discussion of the issues and our major decisions about them, and describes the overall structure of our solution. We then describe in some detail our binding mechanism and our transport level communication protocol. We plan to produce subsequent papers describing our facilities for encryption based security, and providing more information about manufacture of the *stub* modules (which are responsible for interpretation of arguments and results of RPC calls) and our experiences of practical use of this facility.

Environment

The remote procedure call package we have built was developed primarily for use within the Cedar programming environment, communicating across the Xerox research inter-network. When building such a package, some characteristics of the environment inevitably have an impact on the design, so the environment is summarised here.

Cedar [6] is a large project concerned with developing a programming environment that is powerful and convenient for the building of experimental programs and systems. There is an emphasis on uniform, highly interactive user interfaces, and the ease of construction and debugging of programs. Cedar is designed to be used on single-user workstations, although it is also used for the construction of servers (shared computers providing common services, accessible through the communication network).

Most of the computers used for Cedar are *Dorados* [8]. The Dorado is a very powerful machine (for example, a simple Algol-style call and return takes less than 10 microseconds). It is equipped with a 24-bit virtual address space (of 16-bit words) and an 80 megabyte disk. A rough approximation is to think of a Dorado as having the power of an IBM 370/168 processor, dedicated to a single user.

Communication between these computers is typically by means of a 3 megabit per second Ethernet [11]. (Some computers are on a 10 megabit per second Ethernet [7].) Most of the computers running Cedar are on the same Ethernet, but some are on different Ethernets elsewhere in our research inter-network. The inter-network is formed by a large number of 3 megabit and 10 megabit Ethernets (presently about 160) connected by leased telephone and satellite links (at data rates of between 4800 and 56000 bits per second). We envisage that our RPC communication will follow the same pattern as we have experienced with other protocols: most communication is on the local Ethernet (so the much lower data rates of the inter-net links are not an inconvenience to our users), and the Ethernets are not overloaded (we very rarely see offered loads above 40% of the capacity of an Ethernet, and 10% is typical).

The *PUP* family of protocols [3] provides uniform access to any computer on this inter-network. Previous PUP protocols include simple unreliable (but high probability) datagram service, and reliable flow controlled byte streams. Between two computers on the same Ethernet, the lower level raw Ethernet packet format is available.

Essentially all programming is in high level languages. The dominant language is *Mesa* [12] (as modified for the purposes of Cedar), although *Smalltalk* and *InterLisp* are also used. There is no assembly language for Dorados.

Aims

The primary purpose of our RPC project was to make distributed computation easy. Previously, it was observed within our research community that the building of communicating programs was a difficult task, which was undertaken only by members of a select group of communication experts. Even researchers with substantial systems experience found it difficult to acquire the specialised expertise required to build distributed systems with the existing tools. This seemed undesirable. We have available to us a very large, very powerful communication network, numerous powerful computers, and an environment that makes building programs relatively easy. The existing communication mechanisms appeared to be a major factor constraining further development of distributed computing. Our hope is that by providing communication with almost as much ease as local procedure calls, people will be encouraged to build and experiment with distributed applications. Hopefully, RPC removes unnecessary difficulties, leaving only fundamental difficulties of building distributed systems: timing, independent failure of components, and the co-existence of independent execution environments.

We had two secondary aims that we hoped would support our purpose. We wanted to make RPC communication highly efficient (within, say, a factor of five beyond the necessary transmission times of the network). This seems important, lest communication becomes so expensive that the application designers strenuously avoid it. Otherwise, the applications that get developed can become quite distorted by their desire to avoid communicating. Additionally, we feel that it is important to make the semantics of the RPC package as powerful as possible, consistent with the aims of simplicity and efficiency. Otherwise, the gains of a single unified communication paradigm will be lost by requiring application programmers to build extra mechanisms on top of the RPC package. An important issue in the design is how to resolve the tension between powerful semantics and efficiency.

Our final major aim was to provide secure communication with RPC. None of the previously implemented protocols had any provision for protecting the data in transit on our networks. This was true even to the extent that passwords were transmitted as clear-text. Our belief was that research on the protocols and mechanisms for secure communication across an open network had reached a stage such that it was reasonable and desirable for us to include this protection in our package. In addition, very few (if any) distributed systems had previously provided secure end-to-end communication, and it had never been applied to RPC, so the design might provide useful research insights.

Fundamental Decisions

It is not an immediate consequence of our aims that we should use procedure calls as the paradigm for expressing control and data transfers. For example, message passing might be a plausible alternative. It is our belief that the choice between these two would not make a major difference to the problems faced by this design, nor to the solutions adopted. The problems of reliable and efficient transmission of a message and its possible reply are quite similar to the problems encountered for remote procedure calls. The problems of passing arguments and results, and of network security, are essentially unchanged. The over-riding consideration that made us choose procedure calls was that this is the major control and data transfer mechanism embedded in our major language, Mesa.

One might also consider using a more parallel paradigm for our communication, such as some form of remote *fork*. Since our language already includes a construct for forking parallel computations, we could have chosen this as the point at which to add communication semantics. Again, this would not change the major design problems significantly.

We discarded the possibility of emulating some form of shared address space amongst the computers. Previous work has shown that with sufficient care moderate efficiency can be achieved in doing this [14]. We do not know whether an approach employing shared addresses is feasible, but two potentially major difficulties spring to mind: firstly, whether the representation of remote addresses can be integrated into our programming languages (and possibly the underlying machine architecture) without undue upheaval; secondly, whether acceptable efficiency could be achieved. For example, a host in the PUP internet is represented by a 16-bit address, so a naive implementation of a shared address space would extend the width of language addresses by 16-bits. On the other hand, it is possible that careful use of the address mapping mechanisms of our virtual memory hardware could allow shared address space without changing the address width. Even on our 10 megabit Ethernets, the minimum average round trip time for a packet exchange is 120 microseconds [7], so the most likely way to approach this would be to use some form of paging system. In summary, a shared address space between participants in RPC might be feasible, but since we were not willing to undertake that research our subsequent design assumes the absence of shared addresses. Our intuition is that with our hardware the cost of a shared address space would exceed the additional benefits.

A principle that we used several times in making design choices is that the semantics of remote procedure calls should be as close as possible to those of local (single-machine) procedure calls. This principle seems attractive as a way of ensuring that the RPC facility is easy to use, particularly for programmers familiar with single-machine use of our languages and packages. Violation of this principle seemed likely to lead us into the complexities that have made previous communication packages and protocols difficult to use. This principle has occasionally caused us to deviate from designs that would seem attractive to those more experienced in distributed computing. For example, we chose to have no time-out mechanism limiting the duration of a remote call (in the absence of machine or communication failures), whereas most communication packages consider this a worthwhile feature. The argument is that local procedure calls have no time-out mechanism, and our languages include mechanisms to abort an activity as part of the parallel processing mechanism. Designing a new time-out arrangement just for RPC would complicate the programmer's world needlessly. Similarly, we chose the binding semantics described below (based closely on the existing Cedar mechanisms), in preference to the ones presented in Nelson's thesis [13].

Structure

The program structure we use for RPC is similar to that proposed in Nelson's thesis. It is based on the concept of *stubs*. When making a remote call, five pieces of program are involved: the *user*, the *user-stub*, the RPC communications package (known as *RPCRuntime*), the *server-stub*, and the *server*. Their relationship is shown in figure 1. The user, the user-stub, and one instance of RPCRuntime execute in the caller machine; the server, the server-stub and another instance of RPCRuntime execute in the callee machine. When the user wishes to make a remote call, it actually makes a perfectly normal local call which invokes a corresponding procedure in the user-stub. The user-stub is responsible for placing a specification of the target procedure and the arguments into one or more packets and asking the RPCRuntime to transmit these reliably to the callee machine. On receipt of these packets, the RPCRuntime in the callee machine passes them to the server-stub. The server-stub unpacks them and again makes a perfectly normal local call, which invokes the appropriate procedure in the server. Meanwhile, the calling process in the caller machine is suspended awaiting a result packet. When the call in the server completes, it returns to the server-stub and the results are passed back to the suspended process in the caller machine. There they are unpacked and the user-stub returns them to the user. The RPCRuntime is responsible for retransmissions, acknowledgements, packet routing, and encryption. Apart from the effects of multi-machine binding and of machine or communication failures, the call happens just as if the user had invoked the procedure in the server directly. Indeed, if the user and server code were brought into a single machine and bound directly together without the stubs, the program would still work.

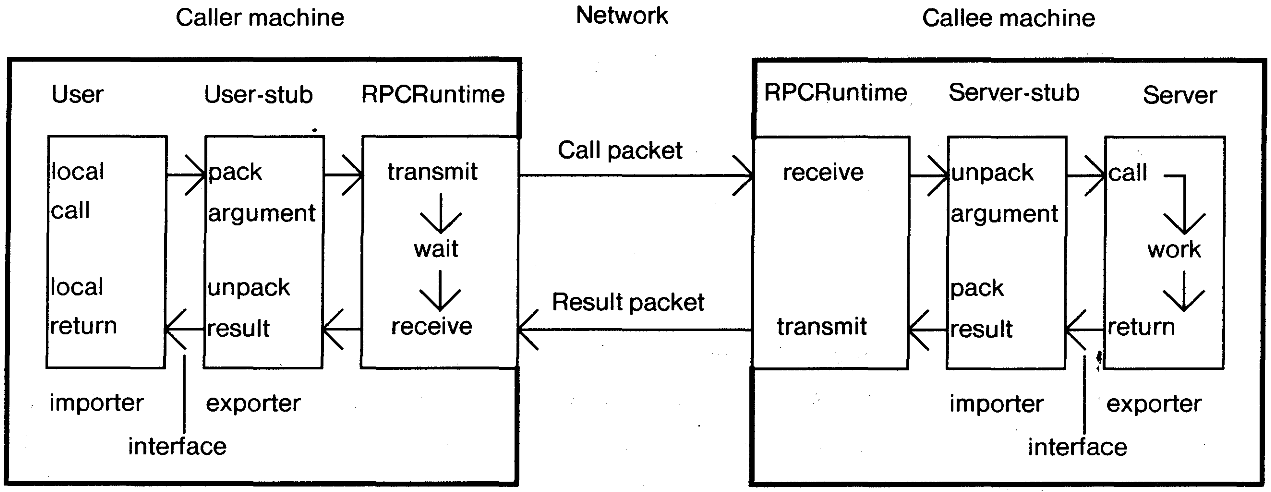

Figure 1: The components of the system, and their interactions for a simple call.

The RPCRuntime is a standard part of the Cedar system. The user and server are written as part of the distributed application. But the user-stub and server-stub are automatically generated, by a program called *Lupine*. This generation is specified by use of Mesa *interface modules*. These are the basis of the Mesa (and Cedar) separate compilation and binding mechanism [9]. An interface module is mainly a list of procedure names, together with the the types of their arguments and results. This is sufficient information for the caller and callee independently to perform compile-time type checking and to generate appropriate calling sequences. A *program module* that implements procedures in an interface is said to *export* that interface. A program module calling procedures from an interface is said to *import* that interface. When writing a distributed application, a programmer first writes an interface module. Then he can write the user code which imports that interface and the server code which exports the interface. He also presents the interface to Lupine, and Lupine will generate the user-stub, (which exports the interface) and the server-stub (which imports the interface). When binding the programs on the caller machine, the user is bound to the user-stub. On the callee machine, the server-stub is bound to the server.

Thus, the programmer does not need to build detailed communication related code. After designing the interface, he need only write the user and server code. Lupine is responsible for generating the code for packing and unpacking arguments and results (and other details of the parameter/result semantics), and for dispatching to the correct procedure for an incoming call in the server-stub. RPCRuntime is responsible for the packet-level communications. The programmer must avoid specifying arguments or results that are incompatible with the lack of shared address space. (Lupine checks this avoidance.) The programmer must also take steps to invoke the inter-machine binding described below, and to handle reported machine or communication failures.

2. Binding

There are two aspects to binding, which we will consider in turn. Firstly, how does a client of the binding mechanism specify what he wants to be bound to? Secondly, how does a caller determine the machine address of the callee and how does the caller specify to the callee which procedure is to be invoked? The first is primarily a question of *naming* and the second a question of *location*.

Naming

The binding operation offered by our RPC package is to bind an importer of an interface to an exporter of an interface. After binding, calls made by the importer invoke procedures implemented by the (remote) exporter. There are two parts to the name of an interface, known as the *type* and the *instance*. The type is intended to specify, at some level of abstraction, which interface the caller expects the callee to implement. The instance is intended to specify which particular implementor of an abstract interface is desired. For example, the type of an interface might correspond to the abstraction of "mail server", and the instance would correspond to some particular mail server selected from many. A reasonable default for the type of an interface might be a name derived from the name of the Mesa interface module. Fundamentally, the semantics of an interface name are not dictated by the RPC package - they are an agreement between the exporter and the importer, not fully enforceable by the RPC package. However, the means by which an exporter uses the interface name to locate an exporter *are* dictated by the RPC package, and these we now describe.

Locating an Appropriate Exporter

We use the Grapevine distributed database [1] for our RPC binding. The major attraction of using Grapevine is that it is widely and reliably available. The Grapevine database is distributed across multiple servers strategically located in our inter-net topology, and Grapevine is configured to maintain at least three copies of each database entry. Since the Grapevine servers themselves are highly reliable and the data is replicated, it is extremely rare for us to be unable to look up a database entry. There are alternatives to using such a database, but we find them unsatisfactory. For example, we could include in our application programs the network addresses of the machine with which they wish to communicate: this binds to a particular machine much too early for most applications. Alternatively, we could use some form of broadcast protocol to locate the desired machine: this is sometimes acceptable, but as a general mechanism it causes too much interference with innocent bystanders, and it is not convenient for binding to machines which are not on the same local network.

Grapevine's database consists of a set of entries each keyed by a character string known as a Grapevine *RName*. There are two varieties of entries: *individuals* and *groups*. Grapevine keeps several items of information for each database entry, but the RPC package is concerned with only two: for each individual there is a *connect-site*, which is a network address, and for each group there is a *member-list*, which is a list of RNames. The RPC package maintains two entries in the Grapevine database for each interface name: one for each type and one for each instance; so the type and instance are both Grapevine RNames. The database entry for the instance is a Grapevine individual whose connect-site is a network address, specifically the network address of the machine on which that instance was last exported. The database entry for the type is a Grapevine group whose members are the Grapevine RNames of the instances of that type which have been exported. For example, if the remote interface with type FileAccess.Alpine and instance Ebbets.Alpine has been exported by a server running at network address 3#22#, and the remote interface with type FileAccess.Alpine and instance Luther.Alpine has been exported by a server running at network address 3#276#, then the members of the Grapevine group FileAccess.Alpine would include Ebbets.Alpine and Luther.Alpine. The Grapevine individual Ebbets.Alpine would have 3#22# as its connect-site and Luther.Alpine would have 3#276#.

When an exporter wishes to make his interface available to remote clients, the server code calls the server-stub which in turn calls a procedure, ExportInterface, in the RPCRuntime. ExportInterface is given the interface name (type and instance) together with a procedure (known as the *dispatcher*) implemented in the server-stub which will handle incoming calls for the interface. ExportInterface calls Grapevine and ensures that the instance is one of the members of the Grapevine group which is the type, and that the connect-site of (the Grapevine individual which is) the instance is the network address of the exporting machine. This may involve updating the database. As an optimisation, the database is not updated if it already contains the correct information - this is usually true: typically an interface of this name has previously been exported, and typically from the same network address. For example, to export the interface with type FileAccess.Alpine and instance Ebbets.Alpine from network address 3#22#, the RPCRuntime would ensure that Ebbets.Alpine in the Grapevine database has connect-site 3#22# and that Ebbets.Alpine is a member of FileAccess.Alpine. The RPCRuntime then records information about this export in a table maintained on the exporting machine. For each currently exported interface, this table contains the interface name, the dispatcher procedure from the server-stub, and a 32-bit value that serves as a permanently unique (machine-relative) identifier of the export. This table is implemented as an array indexed by a small integer. The identifier is guaranteed to be permanently unique by using successive values of a 32-bit counter; on start-up this counter is initialised to a one-second real time clock, and the counter is constrained subsequently to be less than the current value of that clock. This constrains the rate of calls on ExportInterface in a single machine to an *average* rate of less than one per second, averaged over the time since the exporting machine was restarted. The burst rate of such calls can exceed one per second.

When an importer wishes to bind to an exporter, the user code calls its user-stub which in turn calls a procedure, ImportInterface, in the RPCRuntime, giving it the desired interface type and instance. The RPCRuntime determines the network address of the exporter (if there is one) by asking Grapevine for the network address which is the connect-site of the interface instance. The RPCRuntime then makes a remote procedure call to the RPCRuntime package on that machine asking for the binding information associated with this interface type and instance. If the specified machine is not currently exporting that interface this fact is returned to the importing machine and the binding fails. If the specified machine is currently exporting that interface, then the table of current exports maintained by its RPCRuntime yields the corresponding unique identifier; the identifier and the table index are returned to the importing machine and the binding succeeds. The exporter network address, identifier, and table index are remembered by the user-stub for use in remote calls.

Subsequently, when that user-stub is making a call on the imported remote interface, the call packet it manufactures contains the unique identifier and table index of the desired interface, and the entry point number of the desired procedure relative to the interface. When the RPCRuntime on the callee machine receives a new call packet it uses the index to look up its table of current exports (efficiently), verifies that the unique identifier in the packet matches that in the table, and passes the call packet to the dispatcher procedure specified in the table.

There are several variants of this binding scheme available to our clients. If the importer calling ImportInterface specifies only the interface type but no instance, the RPCRuntime obtains from Grapevine the members of the Grapevine group named by the type. The RPCRuntime then obtains the network address for each of those Grapevine individuals, and tries the addresses in turn to find some instance that will accept the binding request; this is done efficiently, and in an order which tends to locate the closest (most responsive) running exporter. This allows an importer to achieve the effect of becoming bound to the closest running instance of a replicated service, in the case where the importer does not care which instance. Of course, an importer is free to enumerate the instances himself, by enumerating the members of the group named by the type.

The instance may be a network address constant instead of a Grapevine name. This allows the importer to bind to the exporter without any interaction with Grapevine, at the cost of including an explicit address in the application programs.

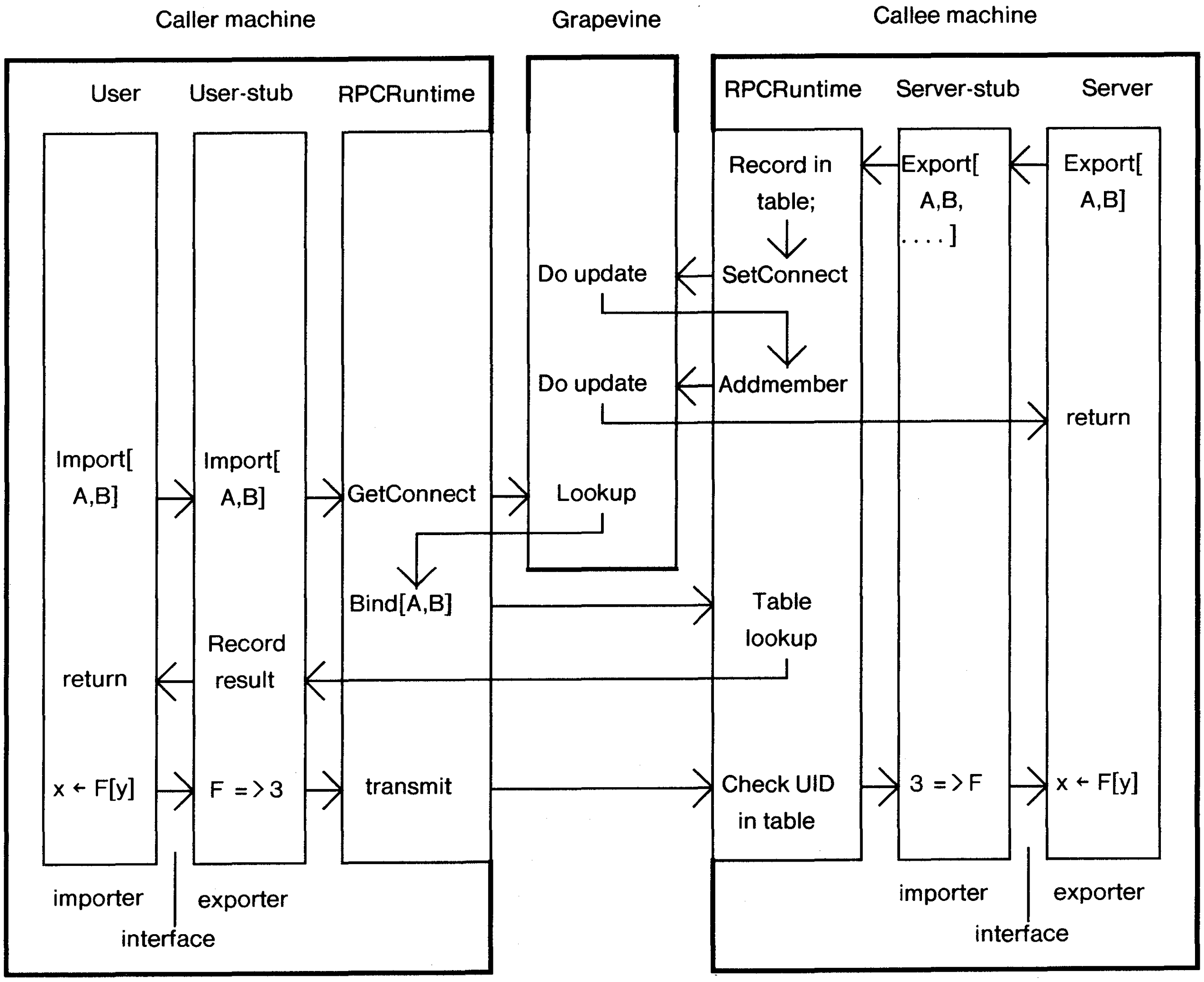

Figure 2: The sequence of events in binding and a subsequent call. The callee machine exports the remote interface with type "A" and instance "B". The caller machine then imports that interface. Then we show the caller initiating a call to procedure "F", which is the third procedure of that interface. The return is not shown.

Discussions

There are some important effects of this scheme. Notice that importing an interface has no effect on the data structures in the exporting machine; this is advantageous when building servers that may have hundreds of users, and avoids problems about what the server should do about this information in relation to subsequent importer crashes. Also the unique identifier scheme means that bindings are implicitly broken if the exporter crashes and restarts (since the currency of the identifier is checked on each call). We believe that this implicit unbinding is the correct semantics - otherwise a user will not be notified of a crash happening between calls. Finally, note that this scheme allows calls to be made only on procedures that have been explicitly exported through the RPC mechanism. An alternative, slightly more efficient scheme, would have been to issue importers with the exporter's internal representation of the server-stub dispatcher procedure; this we considered undesirable since it would allow unchecked access to almost any procedure in the server machine and, therefore, would have made it impossible to enforce any protection or security schemes.

The access controls that restrict updates to the Grapevine database have the effect of restricting who may claim to export particular interface names. This is the desired semantics: it should not be possible, for example, for a random user to claim that his workstation is a mail server and thereby be able to intercept my message traffic. In the case of a replicated service, this access control effect is critical. A client of a replicated service may not know *a priori* the names of the instances of the service. If the client wishes to use two way authentication to get the assurance that the service is genuine, and if we wish to avoid using a single password for identifying every instance of the service, then the client must be able to obtain securely the list of names of the instances of the service. We can achieve this security by employing a secure protocol when the client interacts with Grapevine when importing the interface. Thus, Grapevine's access controls provide the client's assurance that an instance of the service is genuine (authorized).

We have allowed several choices of binding time. The most flexible is where the importer specifies only the type of the interface and not its instance: here the decision about interface instance is made dynamically. Next (and most common) is where the interface instance is a RName, delaying the choice of particular exporting machine. Most restrictive is the facility to specify a network address as instance, thus binding to a particular machine at compile time. We also provide facilities for an importer to dynamically instantiate interfaces and import them. Detailed description of this is too complicated for this paper, but in summary it allows an importer to bind his program to several exporting machines, even when the importer cannot know statically how many machine he wishes to bind to. This has proved to be useful in some open-ended multi-machine algorithms, such as implementing the manager of a distributed atomic transaction. We have not allowed binding at a finer grain than an entire interface. This was not an option we considered, reflecting the absence of use of this mechanism in the packages and systems we have observed.

3. Packet-level Transport Protocol

Requirements

The semantics of RPC can be achieved without designing a specialised packet-level protocol. For example, we could have built our package using the PUP byte stream protocol (or the Xerox NS sequenced packet protocol) as our transport layer. Some of our previous experiments [13] were made using the PUP byte streams, and the Xerox NS "Courier" RPC protocol [4] uses the NS sequenced packet protocol. The Grapevine protocols are essentially similar to remote procedure calls, and use PUP byte streams. Our measurements [13] and experience with each of these implementations convinced us that this approach was unsatisfactory. The particular nature of RPC communication means that there are substantial performance gains available if one designs and implements a transport protocol specially for RPC. Our experiments indicated that a performance gain of a factor of ten might be possible.

An intermediate stance might be tenable: we have never tried the experiment of using an existing transport protocol and building an implementation of it specialized for RPC. However, the request-response nature of communication with RPC is sufficiently different from the large data transfers for which bytes streams are usually employed, that we do not believe this intermediate position.

One aspect we emphasized in our protocol design was minimising the elapsed real-time between initiating a call and getting its results. With protocols for bulk data transfer this is not important: most of the time is spent actually transferring the data. We also strove to minimise the load imposed on a server by substantial numbers of users. When performing bulk data transfers, it is acceptable to adopt schemes that lead to a large cost in setting up and taking down connections, and that require maintenance of substantial state information during a connection. These are acceptable because the costs are likely to be small relative to the data transfer itself. This we believe is untrue for RPC. We envisage our machines being able to serve substantial numbers of clients, and it is unacceptable to require either a large amount of state information or expensive connection handshaking.

It is this level of the RPC package that defines what semantics and guarantees we give for calls. We guarantee that if the call returns to the user then the procedure in the server has been invoked precisely once. Otherwise, an exception is reported to the user and the procedure will have been invoked either once or not at all - the user is not told which. If an exception is reported, the user does not know whether the server has crashed or whether there is a problem in the communication network. Provided the RPCRuntime on the server machine is still responding, there is no upper bound to how long we will wait for results; that is, we will abort a call if there is a communication breakdown or a crash but not if the server code deadlocks or loops. This is the same as the semantics of local procedure calls.

Simple Calls

We have tried to make the per-call communication particularly efficient for the situation where all of the arguments will fit in a single packet buffer, as will all of the results, and where frequent calls are being made. To make a call, the caller sends a *call packet* containing a call identifier (discussed below), data specifying the desired procedure (as described in connection with binding), and the arguments. When the callee machine receives this packet the appropriate procedure is invoked. When the procedure returns, a *result packet* containing the same call identifier, and the results, is sent back to the caller.

The machine that transmits a packet is responsible for retransmitting it until an acknowledgement is received, in order to compensate for lost packets. However, the result of a call is sufficient acknowledgement that the call packet was received, and a call packet is sufficient to acknowledge the result packet of the previous call made by that process. Thus in a situation where the duration of a call and the interval between calls are each less than the retransmission interval, we transmit precisely two packets per call (one in each direction). If the call lasts longer or there is a longer interval between calls up to two additional packets may be sent (the retransmission and an explicit acknowledgement packet); we believe this to be acceptable because in those situations it is clear that communication costs are no longer the limiting factor on performance.

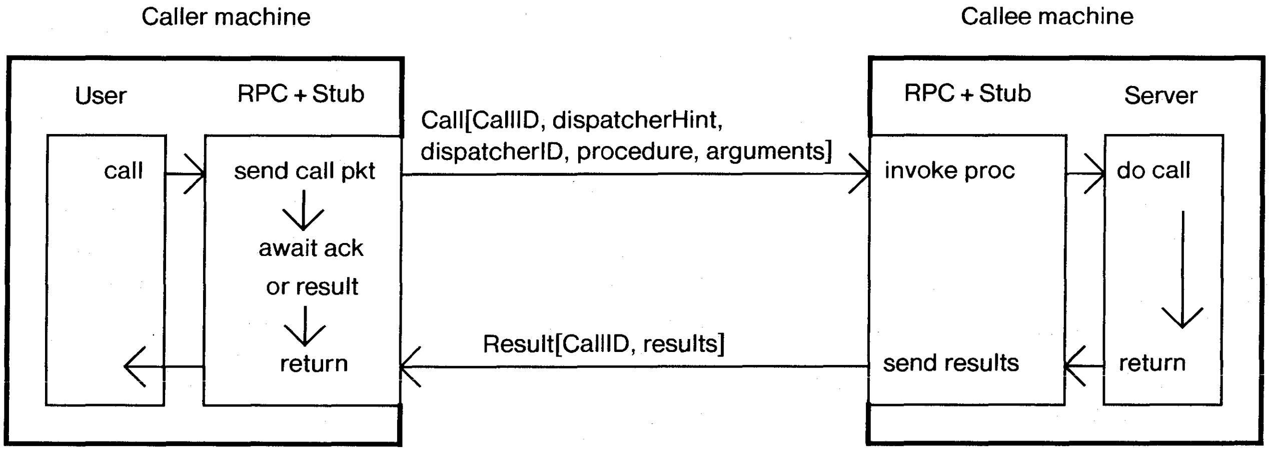

Figure 3: The packets transmitted during a simple call.

The call identifier serves two purposes. It allows the caller to determine that the result packet is truly the result of his current call (not, for example, a much delayed result of some previous call), and it allows the callee to eliminate duplicate call packets (caused by retransmissions, for example). The call identifier consists of the calling machine identifier (which is permanent and globally unique), a machine-relative identifier of the calling process, and a sequence number. We term the pair [machine identifier, process] an *activity*. The important property of an activity is that each activity has at most one outstanding remote call at any time - it will not initiate a new call until it has received the results of the preceding call. The call sequence number must be monotonic for each activity (not necessarily sequential). The RPCRuntime on a callee machine maintains a table giving the sequence number of the last call invoked by each calling activity. When a call packet is received, its call identifier is looked up in this table. The call packet can be discarded as a duplicate (possibly after acknowledgement) unless its sequence number is greater than that given in this table. Figure 3 shows the packets transmitted in simple calls.

It is interesting to compare this arrangement with connection establishment, maintenance and termination in more heavy-weight transport protocols. In our protocol, we think of a *connection* as the shared state information between an activity on a calling machine and the RPCRuntime package on the server machine accepting calls from that activity. We require no special connection establishment protocol (compared with a two packet handshake in many other protocols); receipt of a call packet from a previously unknown activity is sufficient to create the connection implicitly. When the connection is active (that is, there is a call currently being handled, or the last result packet of the call has not yet been acknowledged), both ends maintain significant amounts of state information. However, when the connection is idle the only state information in the server machine is the entry in its table of sequence numbers. A caller has minimal state information when a connection is idle: a single machine-wide counter is sufficient. When initiating a new call, its sequence number is just the next value of this counter. This is why sequence numbers in the calls from an activity are required only to be monotonic, not sequential. When a connection is idle, no process in either machine is concerned with the connection. No communications (such as "pinging" packet exchanges) are required to maintain idle connections. We have no explicit connection termination protocol. If a connection is idle, the server machine may discard its state information after an interval, when there is no longer any danger of receiving retransmitted call packets (say, after five minutes), and it can do so without interacting with the caller machine. This scheme provides the guarantees of traditional connection-oriented protocols without the costs. Note, however, that we rely on the unique identifier we introduced when doing remote binding. Without that identifier we would be unable to detect duplicates if a server crashed then restarted while a caller was still retransmitting a call packet (not very likely, but just plausible). We are also assuming that the call sequence number from an activity does not repeat even if the calling machine is restarted (otherwise a call from the restarted machine might be eliminated as a duplicate). In practice, we achieve this as a side effect of a 32-bit *conversation identifier* which we use in connection with secure calls. For non-secure calls, a conversation identifier may be thought of as a permanently unique identifier which distinguishes incarnations of a calling machine. The conversation identifier is passed with the call sequence number on every call. We generate conversation identifiers based on a 32-bit clock maintained by every machine (initialised from network time servers when a machine restarts).

From experience with previous systems, we anticipate that this light-weight connection management will be important in building large and busy distributed systems.

Complicated Calls

As mentioned above, the transmitter of a packet is responsible for retransmitting it until it is acknowledged. When doing so, the packet is modified to request an explicit acknowledgement. This handles lost packets, long duration calls, and long gaps between calls. When the caller is satisfied with his acknowledgements, the caller process waits for the result packet. While waiting, however, the caller periodically sends a *probe* packet to the callee, which the callee is expected to acknowledge. This allows the caller to notice if the callee has crashed or if there is some serious communication failure, and to notify the user of an exception. Provided these probes continue to be acknowledged the caller will wait indefinitely, happy in the knowledge that the callee is (or claims to be) working on the call. In our implementation the first of these probes is issued after a delay of slightly more than the approximate round-trip time between the machines. The interval between probes increases gradually, until after about ten minutes the probes are being sent once every five minutes. Each probe is subject to retransmission strategies similar to those used for other packets of the call. So if there is a communication failure, the caller will be told about it fairly soon, relative to the total time the caller has been waiting for the result of the call. Note that this will detect only failures in the communication levels: it will not detect if the callee has deadlocked while working on the call. This is in keeping with our principle of making RPC semantics similar to local procedure call semantics. We have language facilities available for watching a process and aborting it if this seems appropriate; these facilities are just as suitable for a process waiting on a remote call.

A possible alternative strategy for retransmissions and acknowledgements is to have the recipient of a packet spontaneously generate an acknowledgement if he doesn't generate the next packet significantly sooner than the expected retransmission interval. This would save the retransmission of a packet when handling long duration calls or large gaps between calls. We decided that saving this packet was not a large enough gain to merit the extra cost of detecting that the spontaneous acknowledgement was needed. In our implementation this extra cost would take the form of maintaining an additional data structure to enable an extra process in the server to generate the spontaneous acknowledgement when appropriate, plus the computational cost of the extra process deciding when to generate the acknowledgement. In particular, it would be difficult to avoid incurring extra cost in the case where the acknowledgement was not needed. There is no analogous extra cost to the caller, since the caller necessarily has a retransmission algorithm in case the call packet is lost.

If the arguments (or results) are too large to fit in a single packet, they are sent in multiple packets with each but the last requesting explicit acknowledgement. Thus when transmitting a large call argument packets are sent alternately by the caller and callee, with the caller sending data packets and the callee responding with acknowledgements. This allows the implementation to use only one packet buffer at each end for the call, and avoids the necessity of including the buffering and flow control strategies found in normal bulk data transfer protocols. To permit duplicate elimination, these multiple data packets within a call each have a call-relative sequence number. Figure 4 shows the packet sequences for complicated calls.

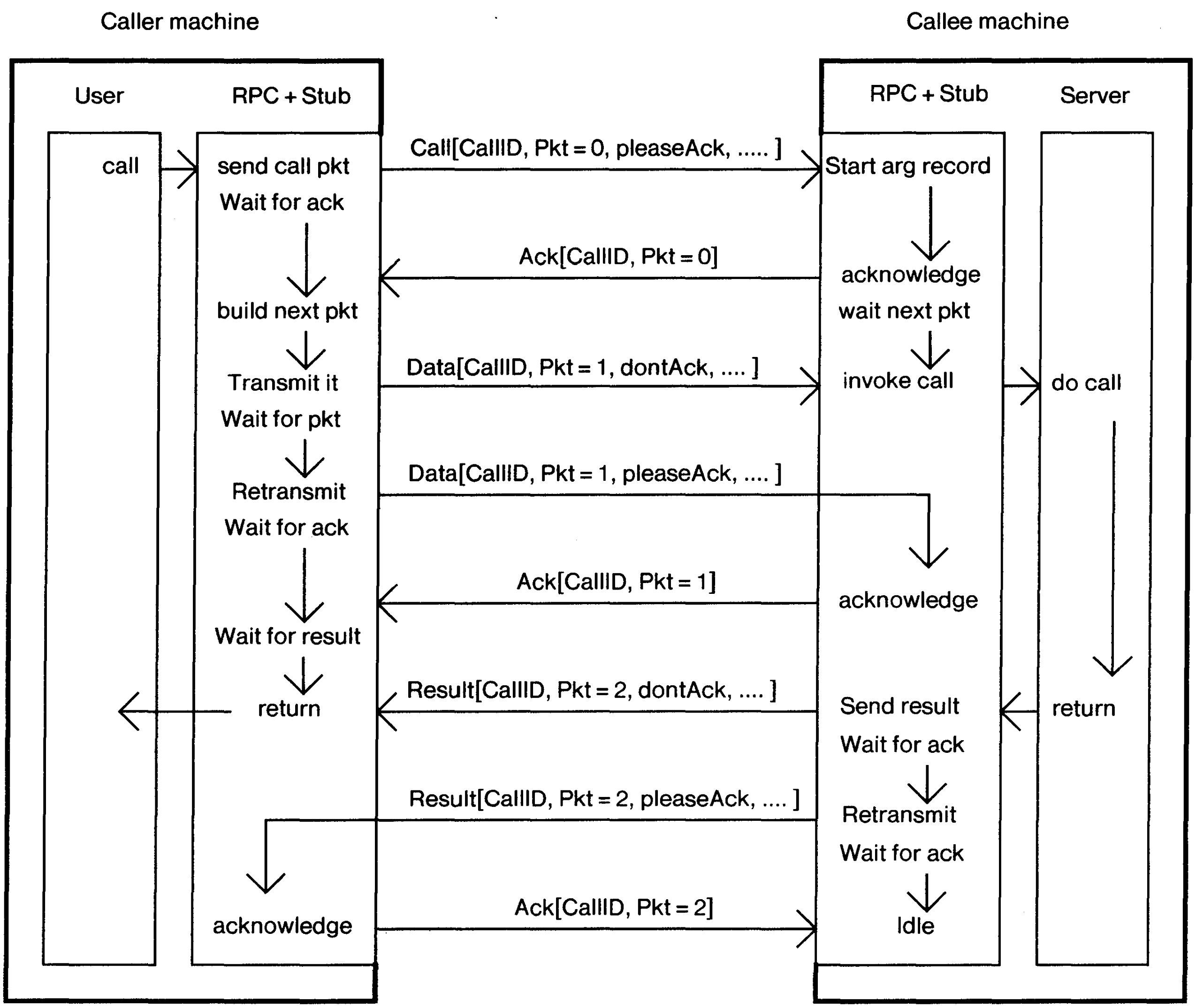

Figure 4: A complicated call. The arguments occupy two packets. The call duration is long enough to require retransmission of the last argument packet requesting an acknowledgement, and the result packet is retransmitted requesting an acknowledgement because no subsequent call arrived.

As described earlier, this protocol concentrates on handling simple calls on local networks. If the call requires more than one packet for its arguments or results, our protocol sends more packets than logically required. We believe this is acceptable; there is still a need for protocols designed for efficient transfer of bulk data, and we have not tried to incorporate both RPC and bulk data in a single protocol. For transferring large amount of data in one direction, our protocol sends up to twice as many packets as a good bulk data protocol would send (since we acknowledge each packet). This would be particularly inappropriate across long haul networks with large delays but high data rates. However, if the communication activity can reasonably be represented as procedure calls, then our protocol has desirable characteristics even across such long haul networks. It is sometimes practical to use our RPC for bulk data transfer across such networks, by multiplexing the data between several processes each making single packet calls - the penalty then is just the extra acknowledgement per packet, and in some situations this is acceptable. The dominant advantage of requiring one acknowledgement for each argument packet (except the last one) is that it simplifies and optimizes the implementation. It would be possible to use our protocol for simple calls, and switch automatically to a more conventional protocol for complicated ones. We have not explored this possibility.

Exception Handling

The Mesa language provides quite elaborate facilities for a procedure to notify exceptions to its caller. These exceptions, called *signals*, may be thought of as dynamically bound procedure activations: when an exception is raised, the Mesa runtime system dynamically scans the call stack to determine if there is a *catch phrase* for the exception. If so, the body of the catch phrase is executed, with arguments given when the exception was raised. The catch phrase may return (with results) causing execution to resume where the exception was raised, or the catch phrase may terminate with a jump out into a lexically enclosing context. In the case of such termination, the dynamically newer procedure activations on the call stack are unwound (in most-recent-first order).

Our RPC package faithfully emulates this mechanism. There are facilities in the protocol to allow the process on the server machine handling a call to transmit an exception packet in place of a result packet. This packet is handled by the RPCRuntime on the caller machine approximately as if it were a call packet, but instead of invoking a new call it raises an exception in the appropriate process. If there is an appropriate catch phrase, it is executed. If the catch phrase returns, the results are passed back to the callee machine, and events then proceed normally. If the catch phrase terminates by a jump then this is notified to the callee machine, where the appropriate procedure activations are unwound. So, we have again emulated the semantics of local calls. This is not quite true: in fact we permit the callee machine to communicate only those exceptions which are defined in the Mesa interface which the callee exported. This simplifies our implementation (in translating the exception names from the callee's machine environment to the caller's), and provides some protection and debugging assistance. The programming convention in single machine programs is that if a package wants to communicate an exception to its caller then the exception should be defined in the package's interface; other exceptions should be handled by a debugger. We have maintained and enforced this convention for RPC exceptions.

In addition to exceptions raised by the callee, the RPCRuntime may raise a *call failed* exception if there is some communication difficulty. This is the primary way in which our clients note the difference between local and remote calls.

Use of Processes

In Mesa and Cedar, parallel processes are available as a built-in language feature. Process creation and changing the processor state on a process swap are generally thought of as being quite cheap. For example, forking a new process costs about as much as ten (local) procedure calls. A process swap involves swapping an evaluation stack and one register, and invalidating some cached information. However, on the scale of a remote procedure call, process creation and process swaps can form a significant cost. This was shown by some of Nelson's experiments [13]. Therefore we took care to keep this cost low when building this package and designing our protocol.

The first step in reducing this cost is to maintain in each machine a stock of idle *server processes* that are willing to handle incoming packets. This means that a call can be handled without incurring the cost of process creation, and without the cost of initialising some of the state of the server process. When a server process is entirely finished dealing with a call, it reverts to its idle state instead of dying. Of course, excess idle server processes kill themselves if they were created in response to a transient peak in the number of RPC calls.

Each packet contains a source and destination *process identifier.* In packets from the caller machine, the source process identifier is the calling process. In packets from the callee machine, the source process identifier is the server process handling the call. During a call, when a process transmits a packet it sets the destination process identifier in the packet from the source process identifier in the preceding packet of the call. If a process is waiting for the next packet in a call, the process notes this fact in a (simple) data structure shared with our Ethernet interrupt handler. When the interrupt handler receives an RPC packet, it looks at the destination process identifier. If the corresponding process on this machine is presently waiting for an RPC packet, then the incoming packet is dispatched directly to that process. Otherwise, the packet is dispatched to an idle server process (which then decides whether the packet is part of a current call requiring an acknowledgement, or is the start of a new call that this server process should handle, or is a duplicate that may be discarded). This means that in most cases an incoming packet is given to the process that wants it with one process swap. (Of course, these arrangements are resilient to being given an incorrect process identifier.) When a calling activity initiates a new call, it attempts to use as destination the identifier of the process that handled the previous call from that activity. This is beneficial, since that process is probably waiting for an acknowledgement of the results of the previous call, and the new call packet will be sufficient acknowledgement. Only a slight performance degradation will result from the caller using a wrong destination process, so a caller maintains only a single destination process for each calling process.

In summary, the normal sequence of events is as follows. A process wishing to make a call manufactures the first packet of the call, guesses a plausible value for the destination process identifier and sets the source to be itself. It then presents the packet to the Ethernet output device and waits for an incoming packet. In the callee machine, the interrupt handler receives the packet and notifies an appropriate server process. The server process handles the packet, then manufactures the response packet. The destination process identifier in this packet will be that of the process waiting in the caller machine. When the response packet arrives in the caller machine, the interrupt handler there passes it directly to the calling process. The calling process now knows the process identifier of the server process, and can use this in subsequent packets of the call, or when initiating a later call.

The effect of this scheme is that in simple calls no processes are created, and there are typically only four processes swaps in each call. Inherently, the minimum possible number of process swaps is two (unless we busy-wait) - we incurred the extra two because incoming packets are handled by an interrupt handler instead of being dispatched to the correct process directly by the device microcode (because we decided not to write specialized microcode).

Other Optimisations

The above discussion has shown some optimisations we have adopted. We use subsequent packets for implicit acknowledgement of previous packets, we have attempted to minimize the costs of maintaining our connections, we have avoided costs of establishing and terminating connections, and we have reduced the number of process switches involved in a call. Some other detailed optimisations also have significant payoff.

When transmitting and receiving RPC packets we bypass the software layers that correspond to the normal layers of a protocol hierarchy. (Actually, we only do so in the case where caller and callee are on the same network - we still use the protocol hierarchy for inter-network routing.) This gives substantial performance gains, but it is in some ways cheating: it is a successful optimisation because only the RPC package uses it. That is, we have modified the network driver software to treat RPC packets as a special case, and that would not be profitable if there were ten special cases. However, our aims imply that RPC *is* a special case: we intend it to become the dominant communication protocol. We believe that the utility of this optimisation is not just an artifact of our particular implementation of the layered protocol hierarchy. Rather, it will always be possible for one particular transport level protocol to improve its performance significantly by by-passing the full generality of the lower layers.

There are reasonable optimisations that we have not used. We could refrain from using the internet packet format for local network communication. We could use specialised packet formats for the simple calls. We could implement special purpose network microcode. We could forbid non-RPC communication. We could save even more process switches by using busy-waits. We avoided these because each is in some way inconvenient, and because we believe we have achieved sufficient efficiency for our purposes. These could probably yield an extra factor of two in our performance.

Security

Our RPC package and protocol include facilities for providing encryption based security for calls. These facilities use Grapevine as an authentication service (or *key distribution center*) and use the federal data encryption standard [5]. Callers are given a guarantee of the identity of the callee, and *vice versa.* We provide full end-to-end encryption of calls and results. The encryption techniques provide protection from eavesdropping (and hide patterns of data), and detect attempts at modification, replay, or creation of calls. Unfortunately, there is insufficient space to describe here the additions and modifications we made to support this mechanism. It will be reported in a later paper.

4. Performance

As we have said already, Nelson's thesis included extensive analysis of several RPC protocols and implementations, and included an examination of the contributory factors to the differing performance characteristics. We will not repeat that information here.

We have made the following measurements of use of our RPC package. The measurements were made for remote calls between two Dorados connected by an Ethernet. The Ethernet had a raw data rate of 2.94 megabits per second. The Dorados were running Cedar. The measurements were made on an Ethernet shared with other users, but the network was lightly loaded (apart from our tests), at 5% to 10% of its capacity. The times are all quoted in microseconds, and were measured by counting Dorado microprocessor cycles and dividing by the known crystal frequency. They are accurate to about 10%. The times are elapsed times: they include time spent waiting for the network and interference from other devices. We were measuring the time from the user program invoking the local procedure exported by the user-stub until the corresponding return from that procedure call. Thus the times include the time spent inside the user-stub, the RPCRuntime on both machines, the server-stub, and the server implementation of the procedures (and transmission times in both directions). The test procedures were all exported to a single interface. We were not using any of our encryption facilities.

We measured individually the elapsed times for 12000 calls on each procedure. The table shows the minimum elapsed time we observed, and the median time. We also present the total packet transmission times for each call (as calculated from the known packet sizes used by our protocol, not directly measured). Finally, we present the elapsed time for making corresponding calls if the user program is bound directly to the server program (*i.e.*, the times when making a purely local call, without any involvement of the RPC package). Hopefully, the time for purely local calls will provide the reader with some calibration of the speed of the Dorado processor and the Mesa language. The times for local calls also indicate what part of the total time is due to the use of RPC.

The first five procedures had respectively 0, 1, 2, 4 and 10 arguments and 0, 1, 2, 4 and 10 results, each argument or result being 16 bits long. The next five procedures all had one argument and one result, each argument or result being an array of size 1, 4, 10, 40 and 100 words respectively. The second last line shows a call on a procedure that raises an exception which the caller resumes. The last line is for the same procedure raising an exception that the caller causes to be unwound.

Procedure	*Minimum*	*Median*	*Transmission*	*Local-only*
0 args/results	1059	1097	131	9
1 arg/result	1070	1105	142	10
2 args/results	1077	1127	152	11
4 args/results	1115	1171	174	12
10 args/results	1222	1278	239	17
1 word array	1069	1111	131	10
4 word array	1106	1153	174	13
10 word array	1214	1250	239	16
40 word array	1643	1695	566	51
100 word array	2915	2926	1219	98
resume exception	2555	2637	284	134
unwind exception	3374	3467	284	196

For transferring large amounts of data in one direction, protocols other than RPC have an advantage, since they can transmit less packets in the other direction. Nevertheless, by interleaving parallel remote calls from multiple processes, we have achieved a data rate of 2 megabits per second transferring between Dorado main memories on the 3 megabit Ethernet. This is equal to the rate achieved by our most highly optimised byte stream implementation (written in BCPL).

We have not measured the cost of exporting or importing an interface. Both of these operations are dominated by the time spent talking to the Grapevine server(s). After locating the exporter machine, calling the exporter to determine the dispatcher identifier uses an RPC call with a few words of data.

5. Status and Discussions

The package as we have described it is fully implemented and in use by Cedar programmers. The entire RPCRuntime package amounts to four Cedar modules (packet exchange, packet sequencing, binding and security), totalling about 2200 lines of source code. Lupine (the stub generator) is substantially larger. Clients are using RPC for several projects. These include the complete communication protocol for *Alpine* (a file server supporting multi-machine transactions), and the control communication for an Ethernet based telephone and audio project. (It has also been used for two network games, providing real-time communication between players on multiple machines.) All our clients have found the package convenient to use, although neither of those projects is yet in full-scale use. Implementations of the protocol have since been made for BCPL, InterLisp, SmallTalk and C.

We are still in the early stages of acquiring experience with the use of RPC, and certainly more work needs to be done. We will have much more confidence in the strength of our design and the appropriateness of RPC when it has been used in earnest by the projects that are now committing to it. There are certain circumstances when RPC seems to be the wrong communication paradigm. These correspond to situations where solutions based on multicasting or broadcasting seem more appropriate [2]. It may be that in a distributed environment there are times when procedure calls (together with our language's parallel processing and co-routine facilities) are not a sufficiently powerful tool, even although there appear to be no such situations in a single machine.

One of our hopes in providing an RPC package with high performance and low cost is that it will encourage the development of new distributed applications that were otherwise infeasible. At present it is hard to justify some of our insistence on good performance, because we lack examples where such performance is important. But our belief is that the examples will come: the present lack is because, historically, distributed communication has been inconvenient and slow. Already we are starting to see distributed algorithms being developed without them being considered a major undertaking; if this trend continues we will have been successful.

A question on which we are still undecided is whether a sufficient level of performance for our RPC aims can be achieved by a general purpose transport protocol whose implementation adopts strategies suitable for RPC as well as ones suitable for bulk data transfer. Certainly, there is no entirely convincing argument that it would be impossible. On the other hand, we have not yet seen it achieved.

We believe the parts of our RPC package that we have discussed are of general interest in several ways. They represent a particular point in the design spectrum of RPC. We believe that we have achieved very good performance without adopting extreme measures, and without sacrificing useful call and parameter semantics. The techniques for managing transport level connections so as to minimize the communication costs and the state that must be maintained by a server are important in our experience of servers dealing with large numbers of users. Our binding semantics are quite powerful, but conceptually simple to a programmer familiar with single machine binding. They were easy and efficient to implement.

References

1. Birrell, A. D., Levin, R., Needham, R. M. and Schroeder, M. D. "Grapevine: an exercise in distributed computing," *Communications of the ACM*, vol. 25, no. 4, pp 260-274, April 1982.
2. Boggs, D. R. *"Internet Broadcasting,"* Ph.D. dissertation, Department of Electrical Engineering, Stanford University, January 1982.
3. Boggs, D. R., Shoch, J. F., Taft, E. A. and Metcalfe, R. M. "PUP: An internetwork architecture," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 612-634, April 1980.
4. "Courier: the remote procedure call protocol," *Xerox System Integration Standard XSIS-038112*, Xerox Corporation, Stamford, Connecticut, December 1981.

5. "Data Encryption Standard," *FIPS publication 46*, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977.
6. Deutsch, L. P. and Taft, E. A. "Requirements for an experimental programming environment," *Technical Report CSL-80-10*, Xerox Palo Alto Research Center, 1980.
7. "The Ethernet: a local area network, data link layer, and physical layer specifications (version 1.0)," Digital Equipment Corporation, Intel Corporation, Xerox Corporation, September 1980.
8. Lampson, B. W. and Pier, K. A. "A processor for a high-performance personal computer," *Proceedings of the 7th IEEE Symposium on Computer Architecture*, pp 146-160, May 1980.
9. Lampson, B. W. and Schmidt, E. E. "Practical use of a polymorphic applicative language," to be presented at POPL, Salt Lake City, Utah, January 1983.
10. Liskov, B. "Primitives for distributed computing," *Operating Systems Review*, vol. 13, no. 5, pp. 33-42, December 1979.
11. Metcalfe, R. M. and Boggs, D. R. "Ethernet: distributed packet switching for local computer networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395-404, July 1976.
12. Mitchell, J. G., Maybury, W. and Sweet, R. "Mesa language manual (Version 5.0)," *Technical Report CSL-79-3*, Xerox Palo Alto Research Center, 1979.
13. Nelson, B. J. "Remote procedure call," *Technical Report CSL-81-9*, Xerox Palo Alto Research Center, 1981.
14. Spector, A. Z. "Performing remote operations efficiently on a local computer network," *Communications of the ACM*, vol. 25, no. 4, pp. 246-259, April 1982.
15. White, J. E. "A high-level framework for network-based resource sharing," *Proceedings of the National Computer Conference*, June 1976.

XEROX

Implementing Remote Procedure Calls

Andrew D. Birrell and Bruce Jay Nelson

CSL-83-7