BM CORPORATION. *Systems Network Architecture—Introduction to Sessions Between Logical Units.* IBM Form No. GC290-1869-1, Oct. 1979.
BM CORPORATION. *SQL/Data System Application Programming.* IBM Form No. SH24-5018, ug. 1981.
INDSAY, B. G., SELINGER, P. G., et al. Notes on distributed databases. In *Distributed Data ases,* (Draffan and Poole, Eds.), Cambridge Univ. Press, Cambridge, U.K. (1980), chap. 10, pp. 47–284. Also available as IBM Research Report RJ2517, San Jose, Calif., July 1979.
INDSAY, B. G. Object naming and catalog management for a distributed database manager. In *roc. Second International Conference on Distributed Computing Systems* (Paris, April 8–10, 981), pp. 31–40. Also available as IBM Research Report RJ2914, San Jose, Calif., August 1980.
ISKOV, B. AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed rograms. In *Proc. Ninth ACM SIGACT-SIGPLAN Symp. on the Principles of Programming anguages* (Albuquerque, N.M., January 25–27, 1982), ACM, New York, pp. 7–19.
OHAN, C. AND LINDSAY B. Efficient commit protocols for the tree of processes model of stributed transactions. In *Proc. Second SIGACT-SIGOPS Symposium on Principles of Distrib- ed Computing* (Montreal, Canada, August 17–19, 1983), ACM, New York, pp. 76–88.
OSS, J. E. Nested transactions: An approach to reliable distributed computing. Ph.D. dissertation, MIT/LCS/TR-260, April 1981.
EDHAM, R. M. AND SCHROEDER, M. D. Using encryption for authentication in large networks computers. *Commun. ACM 21,* 12 (Dec. 1978), 993–998.
ERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7,* 2 (June 82), 187–208.
PEN, D. C. AND YOGEN, K. D. The Clearinghouse: A Decentralized Agent for Locating Named jects in a Distributed Environment. Xerox Tech. Rep. OPD-T8103, Oct. 1981.
URGIS, H., MICHELL, J. AND ISRAEL, J. Issues in the design and use of a distributed file system. *M Operating Systems Review 14,* 3 (July 1980), 55–69.
NDEM COMPUTERS *Transaction Monitoring Facility (TMF) User's Guide,* Oct. 1981.
LLIAMS, R., DANIELS, D., HAAS, L., LOPIS, G., LINDSAY, B., NG, P., OBERMARCK, R., LINGER, P., WALKER, A., WILMS, P., AND YOST, R. R*: An overview of the architecture. In *proving Usability and Responsiveness* (P. Scheurman, Ed.), Academic Press, New York, pp. 27. Also available as IBM Research Report RJ 3325, San Jose, Calif, Dec. 1981.

d March, 1983; revised August 1983; accepted October 1983

Implementing Remote Procedure Calls

ANDREW D. BIRRELL and BRUCE JAY NELSON
Xerox Palo Alto Research Center

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 the 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 optimizations used to achieve high performance and to minimize the load on server machines that have many clients.

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*

General Terms: Design, Experimentation, Performance, Security

Additional Keywords and Phrases: Remote procedure calls, transport layer protocols, distributed naming and binding, inter-process communication, performance of communication protocols.

1. INTRODUCTION

1.1 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 refer to as the *callee*), and the desired procedure is executed there. When the procedure finishes and produces its results, the results are passed backed 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)

Authors' address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

cute (depending on the details of the parallelism of that environment and
implementation).

are many attractive aspects to this idea. One is clean and simple
cs: these should make it easier to build distributed computations, and to
right. Another is efficiency: procedure calls seem simple enough for the
ication to be quite rapid. A third is generality: in single-machine com-
s, procedures are often the most important mechanism for communica-
ween parts of the algorithm.

lea of RPC has been around for many years. It has been discussed in the
terature many times since at least as far back as 1976 [15]. Nelson's
dissertation [13] is an extensive examination of the design possibilities
PC system and has references to much of the previous work on RPC.
, full-scale implementations of RPC have been rarer than paper designs.
recent efforts include *Courier* in the Xerox NS family of protocols [4],
ent work at MIT [10].

aper results from the construction of an RPC facility for the *Cedar*
We felt, because of earlier work (particularly Nelson's thesis and asso-
periments), that we understood the choices the designer of an RPC
ust make. Our task was to make the choices in light of our particular
environment. In practice, we found that several areas were inadequately
d, and we produced a system whose design has several novel aspects.
ues facing the designer of an RPC facility include: the precise semantics
in the presence of machine and communication failures; the semantics
s-containing arguments in the (possible) absence of a shared address
egration of remote calls into existing (or future) programming systems;
how a caller determines the location and identity of the callee); suitable
for transfer of data and control between caller and callee; and how to
lata integrity and security (if desired) in an open communication
In building our RPC package we addressed each of these issues, but it
le to describe all of them in suitable depth in a single paper. This paper
discussion of the issues and our major decisions about them, and
the overall structure of our solution. We also describe in some detail
ng mechanism and our transport level communication protocol. We
oduce subsequent papers describing our facilities for encryption-based
and providing more information about the manufacture of the *stub*
which are responsible for the interpretation of arguments and results
lls) and our experiences with practical use of this facility.

onment

te-procedure-call package we have built was developed primarily for
the Cedar programming environment, communicating across the
earch internetwork. In building such a package, some characteristics of
nment inevitably have an impact on the design, so the environment is
d here.

5] is a large project concerned with developing a programming environ-
is powerful and convenient for the building of experimental programs
s. There is an emphasis on uniform, highly interactive user interfaces,
f 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 (e.g., 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. 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 internetwork. The internetwork consists of a large number of 3-megabyte and 10-megabyte Ethernets (presently about 160) connected by leased telephone and satellite links (at data rates of between 4800 and 56000 bps). We envisage that our RPC communication will follow the pattern we have experienced with other protocols: most communication is on the local Ethernet (so the much lower data rates of the internet links are not an inconvenience to our users), and the Ethernets are not overloaded (we very rarely see offered loads above 40 percent of the capacity of an Ethernet, and 10 percent is typical).

The PUP family of protocols [3] provides uniform access to any computer on this internetwork. 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.

1.3 Aims

The primary purpose of our RPC project was to make distributed computation easy. Previously, it was observed within our research community that the construction of communicating programs was a difficult task, undertaken only by members of a select group of communication experts. Even researchers with substantial systems experience found it difficult to acquire the specialized expertise required to build distributed systems with 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. RPC will, we hope, remove unnecessary difficulties, leaving only the fundamental difficulties of building distributed systems: timing, independent failure of components, and the coexistence 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

yond the necessary transmission times of the network). This seems ant, lest communication become so expensive that application designers ously avoid it. The applications that might otherwise get developed would orted by their desire to avoid communicating. Additionally, we felt that it portant to make the semantics of the RPC package as powerful as possible, t loss of simplicity or efficiency. Otherwise, the gains of a single unified nication paradigm would be lost by requiring application programmers to tra mechanisms on top of the RPC package. An important issue in design ving the tension between powerful semantics and efficiency.

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

damental Decisions

an immediate consequence of our aims that we should use procedure the paradigm for expressing control and data transfers. For example, passing might be a plausible alternative. It is our belief that a choice these alternatives would not make a major difference in the problems this design, nor in the solutions adopted. The problems of reliable and transmission of a message and of its possible reply are quite similar to lems encountered for remote procedure calls. The problems of passing ts and results, and of network security, are essentialy unchanged. The g consideration that made us choose procedure calls was that they were r control and data transfer mechanism imbedded in our major language,

ght also consider using a more parallel paradigm for our communication, me form of remote *fork*. Since our language already includes a construct g parallel computations, we could have chosen this as the point at which mmunication semantics. Again, this would not have changed the major oblems significantly.

carded the possibility of emulating some form of shared address space e computers. Previous work has shown that with sufficient care mod- iency can be achieved in doing this [14]. We do not know whether an employing shared addresses is feasible, but two potentially major s spring to mind: first, whether the representation of remote addresses egrated into our programming languages (and possibly the underlying rchitecture) without undue upheaval; second, whether acceptable effi- be achieved. For example, a host in the PUP internet is represented t address, so a naive implementation of a shared address space would e width of language addresses by 16-bits. On the other hand, it is hat careful use of the address-mapping mechanisms of our virtual ardware 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. Our 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 needlessly complicate the programmer's world. Similarly, we chose the building semantics described below (based closely on the existing Cedar mechanisms) in preference to the ones presented in Nelson's thesis [13].

1.5 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. RPCRuntime is responsible for retransmissions, acknowledgments, packet routing, and encryption. Apart from the effects of multimachine binding and of machine or communication failures, the call happens just as if the user had

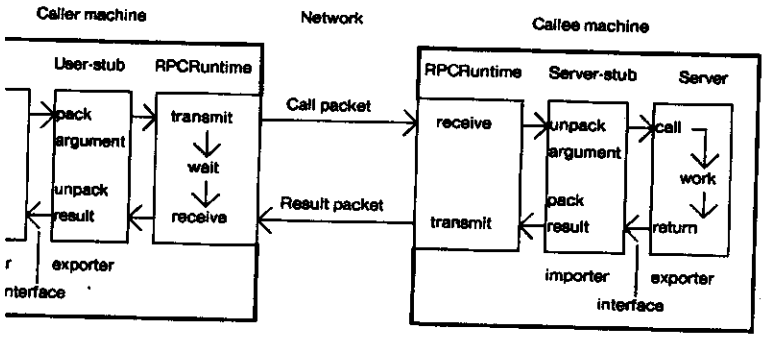

Fig. 1. The components of the system, and their interactions for a simple call.

the procedure in the server directly. Indeed, if the user and server code ought into a single machine and bound directly together without the e program would still work.

untime is a standard part of the Cedar system. The user and server are is part of the distributed application. But the user-stub and server-stub matically generated, by a program called *Lupine*. This generation is by use of Mesa *interface modules*. These are the basis of the Mesa (and eparate compilation and binding mechanism [9]. An interface module is list of procedure names, together with the types of their arguments and his is sufficient information for the caller and callee to independently compile-time type checking and to generate appropriate calling se- A *program module* that implements procedures in an interface is said to at interface. A program module calling procedures from an interface is *port* that interface. When writing a distributed application, a program- writes an interface module. Then he can write the user code that imports rface and the server code that exports the interface. He also presents ace to Lupine, which generates the user-stub, (that exports the interface) erver-stub (that imports the interface). When binding the programs on machine, the user is bound to the user-stub. On the callee machine, -stub is bound to the server.

he programmer does not need to build detailed communication-related r designing the interface, he need only write the user and server code. responsible for generating the code for packing and unpacking argu- d results (and other details of parameter/result semantics), and for g to the correct procedure for an incoming call in the server-stub. ime is responsible for packet-level communications. The programmer d specifying arguments or results that are incompatible with the lack address space. (Lupine checks this avoidance.) The programmer must teps to invoke the intermachine binding described in Section 2, and to orted machine or communication failures.

IG

two aspects to binding which we consider in turn. First, how does a ne binding mechanism specify what he wants to be bound to? Second,

how does a caller determine the machine address of the callee and specify to the callee the procedure to be invoked? The first is primarily a question of *naming* and the second a question of *location*.

2.1 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: 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.

2.2 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. Grapevine is distributed across multiple servers strategically located in our internet topology, and 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 would bind 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 would sometimes be acceptable, but as a general mechanism would cause too much interference with innocent bystanders, and would not be convenient for binding to machines 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

ɜen exported. For example, if the remote interface with type
ss.Alpine and instance Ebbets.Alpine has been exported by a server
at network address 3#22#, and the remote interface with type
ss.Alpine and instance Luther.Alpine has been exported by a server
at network address 3#276#, then the members of the Grapevine
leAccess.Alpine would include Ebbets.Alpine and Luther.Alpine. The
ıe individual Ebbets. Alpine would have 3#22# as its connect-site
er.Alpine would have 3#276#.

an exporter wishes to make his interface available to remote clients, the
ıde calls the server-stub which in turn calls a procedure, ExportInterface,
ɾPCRuntime. ExportInterface is given the interface name (type and
) together with a procedure (known as the *dispatcher*) implemented in
ɾr-stub which will handle incoming calls for the interface. ExportInterface
ıpevine and ensures that the instance is one of the members of the
ıe group which is the type, and that the connect-site of (the Grapevine
ıl which is) the instance is the network address of the exporting machine.
ɣ involve updating the database. As an optimization, the database is not
if it already contains the correct information—this is usually true:
an interface of this name has previously been exported, and typically
same network address. For example, to export the interface with type
ɜs.Alpine and instance Ebbets.Alpine from network address 3#22#, the
ɨtime would ensure that Ebbets.Alpine in the Grapevine database has
ɜite 3#22# and that Ebbets.Alpine is a member of FileAccess.Alpine.
ɔRuntime then records information about this export in a table main-
ı the exporting machine. For each currently exported interface, this table
the interface name, the dispatcher procedure from the server-stub, and
ɣalue that serves as a permanently unique (machine-relative) identifier
port. This table is implemented as an array indexed by a small integer.
tifier is guaranteed to be permanently unique by the use of successive
a 32-bit counter; on start-up this counter is initialized to a one-second
clock, and the counter is constrained subsequently to be less than the
ɣalue of that clock. This constrains the rate of calls on ExportInterface
e machine to an *average* rate of less than one per second, averaged over
since the exporting machine was restarted. The burst rate of such calls
ɜd one per second (see Figure 2).

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

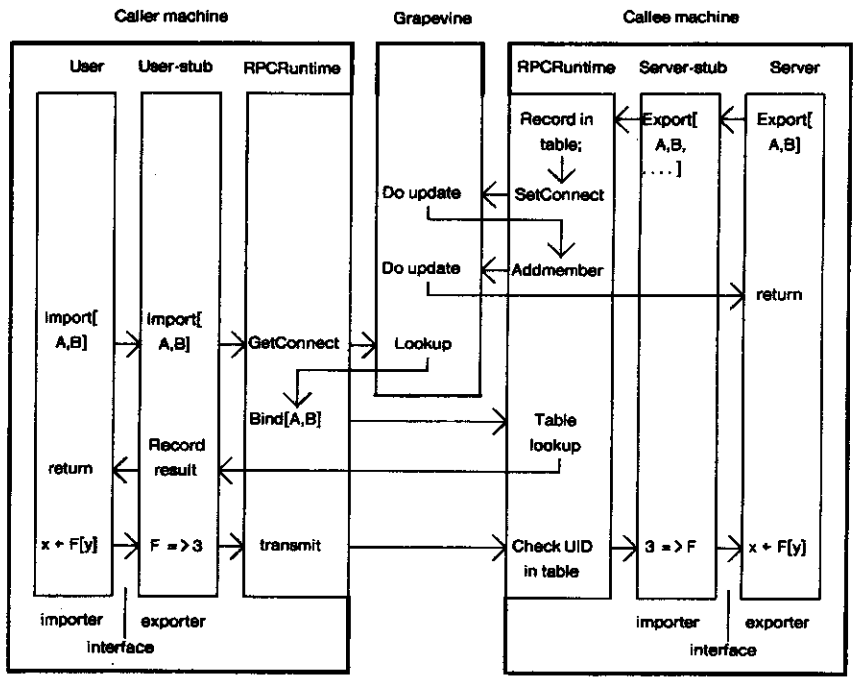

Fig. 2. The sequence of events in binding and a subsequent call. The callee machine exports the remote interface with type A and instance B. The caller machine then imports that interface. We then show the caller initiating a call to procedure F, which is the third procedure of that interface. The return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

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,

an order which tends to locate the closest (most responsive) running
:. This allows an importer to become bound to the closest running instance
licated service, where the importer does not care which instance. Of
an importer is free to enumerate the instances himself, by enumerating
ibers of the group named by the type.
istance may be a network address constant instead of a Grapevine name.
uld allow the importer to bind to the exporter without any interaction
apevine, at the cost of including an explicit address in the application
is.

cussion

re some important effects of this scheme. Notice that importing an
e has no effect on the data structures in the exporting machine; this is
geous when building servers that may have hundreds of users, and avoids
s regarding what the server should do about this information in relation
quent importer crashes. Also, use of the unique identifier scheme means
dings are implicitly broken if the exporter crashes and restarts (since the
r of the identifier is checked on each call). We believe that this implicit
ig is the correct semantics: otherwise a user will not be notified of a
ppening between calls. Finally, note that this scheme allows calls to be
ly on procedures that have been explicitly exported through the RPC
sm. An alternate, slightly more efficient scheme would be to issue
s with the exporter's internal representation of the server-stub dis-
procedure; this we considered undesirable since it would allow unchecked
almost any procedure in the server machine and, therefore, would make
sible to enforce any protection or security schemes.
cess controls that restrict updates to the Grapevine database have the
restricting the set of users who will be able to export particular interface
These are the desired semantics: it should not be possible, for example,
dom user to claim that his workstation is a mail server and to thereby
o intercept my message traffic. In the case of a replicated service, this
ntrol effect is critical. A client of a replicated service may not know a
e names of the instances of the service. If the client wishes to use two-
entication to get the assurance that the service is genuine, and if we
void using a single password for identifying every instance of the service,
client must be able to securely obtain the list of names of the instances
rvice. We can achieve this security by employing a secure protocol when
it interacts with Grapevine as the interface is being imported. Thus
ie's access controls provide the client's assurance that an instance of the
genuine (authorized).
ve allowed several choices for binding time. The most flexible is where
rter specifies only the type of the interface and not its instance: here
ion about the interface instance is made dynamically. Next (and most
is where the interface instance is an RName, delaying the choice of a
r exporting machine. Most restrictive is the facility to specify a network
is an instance, thus binding it to a particular machine at compile time.
provide facilities allowing an importer to dynamically instantiate inter-
l to import them. A detailed description of how this is done would be

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 machines he wishes to bind to. This has proved to be useful in some open-ended multimachine 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, in light of inutility of this mechanism in the packages and systems we have observed.

3. PACKET-LEVEL TRANSPORT PROTOCOL

3.1 Requirements

The semantics of RPCs can be achieved without designing a specialized 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 PUP byte streams, and the Xerox NS "Courier" RPC protocol [4] uses the NS sequenced packet protocol. 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 unlike the large data transfers for which bytes streams are usually employed that we do not believe this intermediate position to be tenable.

One aim we emphasized in our protocol design was minimizing the elapsed real-time between initiating a call and getting 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 minimize 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 for 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 would be unacceptable to require either a large amount of state information or expensive connection handshaking.

It is this level of the RPC package that defines the semantics and the 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

›nding, there is no upper bound on how long we will wait for results; ɛ will abort a call if there is a communication breakdown or a crash but server code deadlocks or loops. This is identical to the semantics of edure calls.

ⅈe Calls

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

ıchine that transmits a packet is responsible for retransmitting it until vledgment is received, in order to compensate for lost packets. However, of a call is sufficient acknowledgment that the call packet was received, l packet is sufficient to acknowledge the result packet of the previous by that process. Thus in a situation where the duration of a call and ⁄al between calls are each less than the transmission interval, we precisely two packets per ćall (one in each direction). If the call lasts there is a longer interval between calls, up to two additional packets ɜnt (the retransmission and an explicit acknowledgment packet); we is to be acceptable because in those situations it is clear that commu- osts are no longer the limiting factor on performance.

l identifier serves two purposes. It allows the caller to determine that packet is truly the result of his current call (not, for example, a much sult of some previous call), and it allows the callee to eliminate duplicate ts (caused by retransmissions, for example). The call identifier consists ling machine identifier (which is permanent and globally unique), a ɛlative identifier of the calling process, and a sequence number. We ıair [machine identifier, process] an *activity*. The important property of ⁄ is that each activity has at most one outstanding remote call at any vill not initiate a new call until it has received the results of the call. The call sequence number must be monotonic for each activity ɛcessarily sequential). The RPCRuntime on a callee machine maintains ving the sequence number of the last call invoked by each calling When a call packet is received, its call identifier is looked up in this call packet can be discarded as a duplicate (possibly after acknowledg- ɛss its sequence number is greater than that given in this table. Figure ıe packets transmitted in simple calls.

ɛresting to compare this arrangement with connection establishment, ıce and termination in more heavyweight transport protocols. In our ve think of a *connection* as the shared state information between an a calling machine and the RPCRuntime package on the server machine ɜalls from that activity. We require no special connection establishment

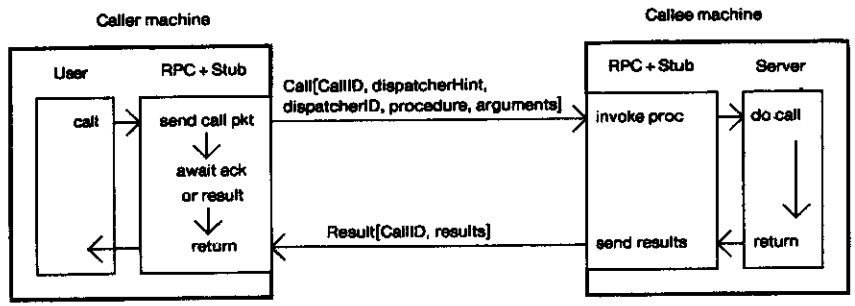

Fig. 3. The packets transmitted during a simple call.

protocol (compared with the two-packet handshake required 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 (when there is a call being handled, or when 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 this identifier we would be unable to detect duplicates if a server crashed and 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 (initialized 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.

nplicated Calls

tioned above, the transmitter of a packet is responsible for retransmitting it is acknowledged. In doing so, the packet is modified to request an acknowledgment. This handles lost packets, long duration calls, and long tween calls. When the caller is satisfied with its acknowledgments, the rocess waits for the result packet. While waiting, however, the caller ally sends a *probe* packet to the callee, which the callee is expected to ledge. This allows the caller to notice if the callee has crashed or if there serious communication failure, and to notify the user of an exception. d these probes continue to be acknowledged the caller will wait indefi- ıappy in the knowledge that the callee is (or claims to be) working on the our implementation the first of these probes is issued after a delay of more than the approximate round-trip time between the machines. The between probes increases gradually, until, after about 10 minutes, the ıre being sent once every five minutes. Each probe is subject to retrans- strategies similar to those used for other packets of the call. So if there munication failure, the caller will be told about it fairly soon, relative to l time the caller has been waiting for the result of the call. Note that this y detect failures in the communication levels: it will not detect if the as deadlocked while working on the call. This is in keeping with our e of making RPC semantics similar to local procedure call semantics. We ıguage facilities available for watching a process and aborting it if this ppropriate; these facilities are just as suitable for a process waiting on a :all.

sible alternative strategy for retransmissions and acknowledgments is to : recipient of a packet spontaneously generate an acknowledgment if he generate the next packet significantly sooner than the expected retrans- interval. This would save the retransmission of a packet when dealing g duration calls or large gaps between calls. We decided that saving this vas not a large enough gain to merit the extra cost of detecting that the eous acknowledgment was needed. In our implementation this extra cost e in the form of maintaining an additional data structure to enable an ocess in the server to generate the spontaneous acknowledgment, when ate, plus the computational cost of the extra process deciding when to the acknowledgment. In particular, it would be difficult to avoid incur- :a cost when the acknowledgment is not needed. There is no analogous it to the caller, since the caller necessarily has a retransmission algorithm he call packet is lost.

arguments (or results) are too large to fit in a single packet, they are nultiple packets with each but the last requesting explicit acknowledg- ıus when transmitting a large call argument packets are sent alternately :aller and callee, with the caller sending data packets and the callee ng with acknowledgments. This allows the implementation to use only :et buffer at each end for the call, and avoids the necessity of including ering and flow control strategies found in normal-bulk data transfer s. To permit duplicate elimination, these multiple data packets within a has a call-relative sequence number. Figure 4 shows the packet sequences licated calls.

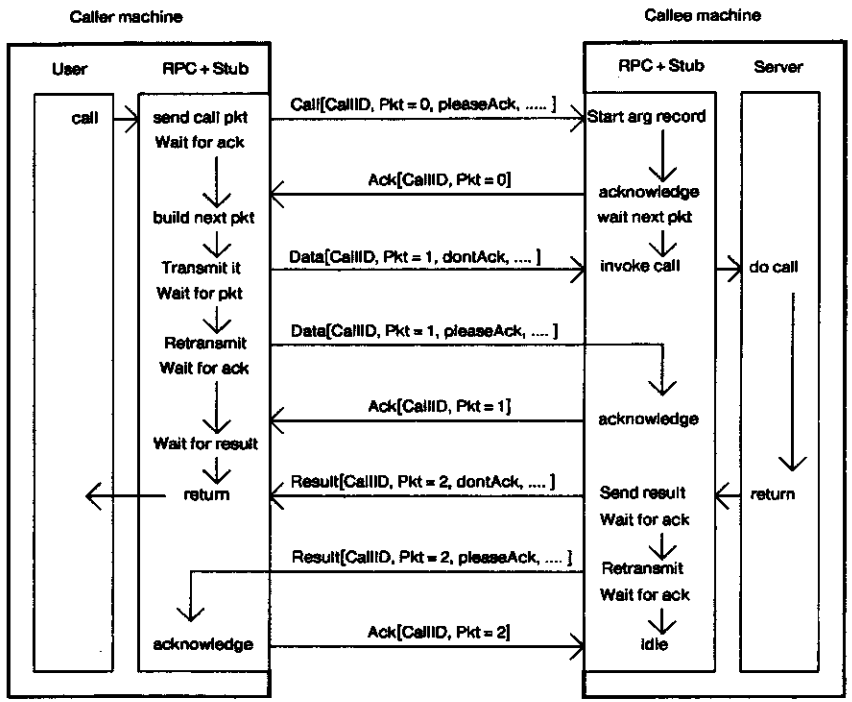

Fig. 4. A complicated call. The arguments occupy two packets. The call duration is long enough to require retransmission of the last argument packet requesting an acknowledgment, and the result packet is retransmitted requesting an acknowledgment because no subsequent call arrived.

As described in Section 3.1, 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 are 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 a 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 and 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 RPC for bulk data transfer across such networks, by multiplexing the data between several processes each of which is making single packet calls—the penalty then is just the extra acknowledgment per packet, and in some situations this is acceptable. The dominant advantage of requiring one acknowledgment for each argument packet (except the last one) is that it simplifies and optimizes the implementation. It would be possible to

r protocol for simple calls, and to switch automatically to a more conven-
protocol for complicated ones. We have not explored this possibility.

ception Handling

esa language provides quite elaborate facilities for a procedure to notify
ons to its caller. These exceptions, called *signals*, may be thought of as
ically bound procedure activations: when an exception is raised, the Mesa
e system dynamically scans the call stack to determine if there is a *catch*
for the exception. If so, the body of the catch phrase is executed, with
nts given when the exception was raised. The catch phrase may return
esults) causing execution to resume where the exception was raised, or the
hrase may terminate with a jump out into a lexically enclosing context.
case of such termination, the dynamically newer procedure activations on
l stack are unwound (in most-recent-first order).
RPC package faithfully emulates this mechanism. There are facilities in
otocol to allow the process on the server machine handling a call to
it an exception packet in place of a result packet. This packet is handled
RPCRuntime on the caller machine approximately as if it were a call
, but instead of invoking a new call it raises an exception in the appropriate
. If there is an appropriate catch phrase, it is executed. If the catch phrase
, the results are passed back to the callee machine, and events proceed
ly. If the catch phrase terminates by a jump then the callee machine is so
l, which then unwinds the appropriate procedure activations. Thus we
gain emulated the semantics of local calls. This is not *quite* true: in fact
mit the callee machine to communicate only those exceptions which are
l in the Mesa interface which the callee exported. This simplifies our
ıentation (in translating the exception names from the callee's machine
ıment to the caller's), and provides some protection and debugging assist-
The programming convention in single machine programs is that if a
e wants to communicate an exception to its caller then the exception
be defined in the package's interface; other exceptions should be handled
ebugger. We have maintained and enforced this convention for RPC
ons.
ddition to exceptions raised by the callee, the RPCRuntime may raise a
ed exception if there is some communication difficulty. This is the primary
which our clients note the difference between local and remote calls.

e of Processes

a and Cedar, parallel processes are available as a built-in language feature.
s creation and changing the processor state on a process swap are consid-
expensive. For example, forking a new process costs about as much as ten
procedure calls. A process swap involves swapping an evaluation stack
ıe register, and invalidating some cached information. However, on the
f a remote procedure call, process creation and process swaps can amount
gnificant cost. This was shown by some of Nelson's experiments [13].
ore we took care to keep this cost low when building this package and
ng our protocol.

The first step in reducing cost is maintaining in each machine a stock of idle *server processes* 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 initializing some of the state of the server process. When a server process is entirely finished 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 *process identifier* for both source and destination. 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 at this time 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 acknowledgment, the start of a new call that this server process should handle, or 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 its destination the identifer of the process that handled the previous call from that activity. This is beneficial, since that process is probably waiting for an acknowledgment of the results of the previous call, and the new call packet will be sufficient acknowledgment. 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 process 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).

ıer Optimizations

bove discussion shows some optimizations we have adopted: we use uent packets for implicit acknowledgment of previous packets, we attempt imize the costs of maintaining our connections, we avoid costs of estab- and terminating connections, and we reduce the number of process es involved in a call. Some other detailed optimizations also have signifi- ıyoff.

ın transmitting and receiving RPC packets we bypass the software layers ırrespond to the normal layers of a protocol hierarchy. (Actually, we only n cases where caller and callee are on the same network—we still use the ol hierarchy for internetwork routing.) This provides substantial perform- ains, but is, in a sense, cheating: it is a successful optimization because ıe RPC package uses it. That is, we have modified the network-driver- :e to treat RPC packets as a special case; this would not be profitable if 'ere ten special cases. However, our aims imply that RPC *is* a special case: ınd it to become the dominant communication protocol. We believe that ity of this optimization is not just an artifact of our particular implemen- ɔf the layered protocol hierarchy. Rather, it will always be possible for one lar transport level protocol to improve its performance significantly by ıing the full generality of the lower layers.

e are reasonable optimizations that we do not use: we could refrain from ıe internet packet format for local network communication, we could use zed packet formats for the simple calls, we could implement special ı network microcode, we could forbid non-RPC communication, or we ave even more process switches by using busy-waits. We have avoided ptimizations because each is in some way inconvenient, and because we we have achieved sufficient efficiency for our purposes. Using them would y have provided an extra factor of two in our performance.

ɔurity

'C package and protocol include facilities for providing encryption-based ' for calls. These facilities use Grapevine as an authentication service (or *ribution center*) and use the federal data encryption standard [5]. Callers n a guarantee of the identity of the callee, and vice versa. We provide full ɔnd encryption of calls and results. The encryption techniques provide on from eavesdropping (and conceal patterns of data), and detect at- at modification, replay, or creation of calls. Unfortunately, there is ient space to describe here the additions and modifications we have made ɔrt this mechanism. It will be reported in a later paper.

=ORMANCE

ıave mentioned already, Nelson's thesis included extensive analysis of RPC protocols and implementations, and included an examination of the ıting factors to the differing performance characteristics. We do not hat information here.

ave made the following measurements of use of our RPC package. The ments were made for remote calls between two Dorados connected by an

Table I. Performance Results for Some Examples of Remote Calls

Procedure	Minimum	Median	Transmission	Local-only
no 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 except'n	2555	2637	284	134
unwind except'n	3374	3467	284	196

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 five to ten percent of capacity. The times shown in Table I are all in microseconds, and were measured by counting Dorado microprocessor cycles and dividing by the known crystal frequency. They are accurate to within about ten percent. The times are elapsed times: they include time spent waiting for the network and time used by interference from other devices. We are measuring from when the user program invokes the local procedure exported by the user-stub until the corresponding return from that procedure call. This interval includes 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 12,000 calls on each procedure. Table I 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, rather than from direct measurement). Finally, we present the elapsed time for making corresponding calls if the user program is bound directly to the server program (i.e., when making a purely local call, without any involvement of the RPC package). The time for purely local calls should 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 line from the bottom 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.

For transferring large amounts of data in one direction, protocols other than RPC have an advantage, since they can transmit fewer packets in the other

ion. Nevertheless, by interleaving parallel remote calls from multiple proc- we have achieved a data rate of 2 megabits per second transferring between lo main memories on the 3 megabit Ethernet. This is equal to the rate ved by our most highly optimized byte stream implementation (written in).

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

ATUS AND DISCUSSIONS

ackage as we have described it is fully implemented and in use by Cedar mmers. The entire RPCRuntime package amounts to four Cedar modules t exchange, packet sequencing, binding and security), totalling about 2,200 f source code. Lupine (the stub generator) is substantially larger. Clients ing RPC for several projects, including the complete communication ol for *Alpine* (a file server supporting multimachine transactions), and the l communication for an Ethernet-based telephone and audio project. (It o been used for two network games, providing real-time communication n players on multiple machines.) All of our clients have found the package ient to use, although neither of the projects is yet in full-scale use. nentations of the protocol have been made for BCPL, InterLisp, SmallTalk

are still in the early stages of acquiring experience with the use of RPC rtainly more work needs to be done. We will have much more confidence strength of our design and the appropriateness of RPC when it has been earnest by the projects that are now committing to it. There are certain stances in which RPC seems to be the wrong communication paradigm. correspond to situations where solutions based on multicasting or broad- seem more appropriate [2]. It may be that in a distributed environment re times when procedure calls (together with our language's parallel ing and coroutine facilities) are not a sufficiently powerful tool, even there do not appear to be any such situations in a single machine.

of our hopes in providing an RPC package with high performance and low that it will encourage the development of new distributed applications re formerly infeasible. At present it is hard to justify some of our insistence l performance because we lack examples demonstrating the importance of rformance. But our belief is that the examples will come: the present lack o the fact that, historically, distributed communication has been incon- and slow. Already we are starting to see distributed algorithms being ed that are not considered a major undertaking; if this trend continues have been successful.

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

We believe the parts of our RPC package here 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 for 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. *Commun. ACM 25*, 4 (April 1982), 260–274.
2. BOGGS, D. R. Internet Broadcasting. PhD dissertation, Department of Electrical Engineering, Stanford University, Jan. 1982.
3. BOGGS, D. R., SHOCH, J. R., TAFT, E. A. AND METCALF, R. M. PUP: An internetwork architecture. *IEEE Trans. Commun. 28*, 4 (April 1980), 612–634.
4. Courier: the remote procedure call protocol. Xerox System Integration Standard XSIS-038112, Xerox Corporation, Stamford, Connecticut, Dec. 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 exceptional programming environment. Tech. Rep. CSL-80-10, Xerox Palo Alto Research Center, Palo Alto, Calif., 1980.
7. Ethernet, a local area network: data link layer and physical layer specifications version 1.0. Digital Equipment Corporation, Intel Corporation, Xerox Corporation, Sept. 1980.
8. LAMPSON, B. W. AND PIER, K. A. A processor for a high-performance personal computer. In *Proc 7th IEEE Symposium on Computer Architecture*, (May 1980), IEEE, New York, pp. 146–160.
9. LAMPSON, B. W. AND SCHMIDT, E. E. Practical use of a polymorphic applicative language. In *Proc. Tenth Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 24–26), ACM, New York (1983), pp. 237–255.
10. LISKOV, B. Primitives for distributed computing. *Oper. Syst. Rev. 13*, 5 (Dec. 1979), 33–42.
11. METCALFE, R. M. AND BOGGS, D. R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July 1976), 395–404.
12. MITCHELL, J. G., MAYBURY, W. AND SWEET, R. Mesa language manual (Version 5.0). Tech. Rep. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif. 1979.
13. NELSON, B. J. Remote procedure call. Tech. Rep. CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, Calif. 1981.
14. SPECTOR, A. Z. Performing remote operations efficiently on a local computer network. *Commun. ACM 25*, 4 (April 1982), 246–260.
15. WHITE, J. E. A high-level framework for network-based resource sharing. In *Proc. National Computer Conference*, (June 1976).

Received March 1983; revised November 1983; accepted November 1983