

Parallel Communication in a Large Distributed Environment

MAHADEV SATYANARAYANAN, MEMBER, IEEE, AND ELLEN H. SIEGEL

Abstract—In this paper, we describe the evolution of *MultiRPC*, a parallel remote procedure call mechanism implemented in Unix. Parallelism is obtained from the concurrency of processing on servers and from the overlapping of retransmissions and timeouts. Each of the parallel calls retains the semantics and functionality of our standard remote procedure calls. The underlying communication medium need not support multicast or broadcast transmissions. We derive an analytic model of the system and validate it. Our experimental observations demonstrate the feasibility of using *MultiRPC* to contact up to 100 servers in parallel.

Index Terms—Andrew, Multicast, *MultiRPC*, network protocols, parallel communication, performance, remote procedure call, *RPC2*.

I. INTRODUCTION

ANDREW is a distributed computing environment that is expected to span over 5000 workstations at Carnegie Mellon University [13]. *RPC2* [17] is a remote procedure call (RPC) mechanism that has been used extensively in Andrew. *MultiRPC* is an extension to *RPC2* that enables a client to invoke multiple remote servers while retaining the control flow and delivery semantics of RPC. In this paper, we establish the importance of parallel RPC, and show that the *MultiRPC* implementation is versatile and simple to use. Furthermore, we present experimental results which demonstrate that the performance of this implementation approaches an analytically derived bound for our operating system environment.

Sections II and III present the rationale and design considerations for *MultiRPC*. Sections IV and V present an overview of *RPC2* and describe how *MultiRPC* extends it. The latter section also explores some of the subtle consequences of our original design decisions, and describes some resulting modifications. Section VI evaluates the system by deriving an analytic performance model, validating it, and exploring the behavior of the system under conditions beyond the range of the model. Section VII describes related work and Section VIII concludes the paper with an overview of work in progress.

Manuscript received July 6, 1987; revised March 10, 1988. This work was supported by the National Science Foundation Contract CCR-8657907, Defense Advanced Research Projects Agency (Order No. 4976, Contract F33615-84-K-1520), and the IBM Corporation. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies of the funding agencies or Carnegie Mellon University.

The authors are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8932909.

II. MOTIVATION

The principles underlying *MultiRPC* arose as a solution to a specific problem in Andrew. In the Andrew file system [16], workstations fetch files from servers and cache them on their local disks. In order to maintain the consistency of the caches, servers maintain *callback* state about the files cached by workstations. A callback on a file is essentially a commitment by a server to a workstation that it will notify the latter of any change to the file. This guarantee maintains consistency while allowing workstations to use cached data without contacting the server on each access. Before a file may be modified on the server, every workstation that has a callback on the file must be notified. Since the system is ultimately expected to encompass over 5000 workstations, an update to a popular file may involve a callback RPC to hundreds or thousands of workstations. The problem is exacerbated by the fact that a callback RPC to a dead or unreachable workstation must time out before the connection is declared broken and the next workstation tried. Each such workstation would cause a delay of many seconds, rather than the few tens of milliseconds typical of RPC roundtrip times for simple requests. Given these observations, we felt that the potential delay in updating widely-cached files would be unacceptable if we were restricted to using simple RPC calls iteratively.

A simple broadcast of callback information is not advisable. With broadcast, every time a file is changed anywhere in the system every workstation would have to process a callback packet and determine if the specified files were relevant to that workstation. Using multicast to narrow the set of workstations contacted is also impractical, because each cached file would have to correspond to a distinct multicast address. Since workstations flush and replace cache entries frequently, the membership of multicast groups would be highly dynamic and difficult to maintain in a consistent manner.

The characteristics of the Andrew networking environment are a further impediment to the use of multicast. The network is composed of diverse physical segments, many of which do not support multicast communication. Furthermore, most of our workstations do not possess appropriate network interface hardware; multicast reception on such workstations has to be simulated by software filtering of broadcast packets. Consequently true multicast is an illusion in our environment at the present time.

Besides these considerations, the use of broadcast or multicast alone does not provide servers with confirmation that individual workstations have indeed received callback information. Such confirmation is implicit in the reliable delivery *seman-*

tics of RPC. It became clear to us that we needed a mechanism that retained strict RPC semantics while overlapping the computation and communication overheads at each of the destinations. This is the essence of MuhiRPC.

Limited degrees of parallelism may be attained by creating multiple threads of control and making a normal RPC call on each thread. However, this is not a solution that scales well. If hundreds of sites were being contacted, the time to create and destroy these threads, the context switching overheads, and the associated stack and control memory usage would be excessive in all commonly used operating systems.

Finally, although MuhiRPC was motivated by callback, it can be of value in other contexts.

- 1 Replication algorithms such as quorum consensus [9] require multiple network sites to be contacted in order to perform an operation. The request to each site is usually the same, although the returned information may be different. MultiRPC could be used to considerably enhance the performance of such algorithms.
- 1 We have used MultiRPC as the key element of a tool to run benchmarks for performance evaluation of Andrew. This tool facilitates the control, coordination, and data collection of many workstations that are participating in the benchmark.
- 1 The performance of some relatively simple but frequent operations in large distributed systems may also be improved by MultiRPC. For example, consider, the contacting of a name or time server. If more than one such server is available, one can use MultiRPC to contact many of them, wait for the earliest reply, and abandon all further replies.
- 1 In a mail system such as Grapevine [1], many mail servers may be contacted to retrieve deposited messages. MuhiRPC could be used to contact these sites in parallel.

III. DESIGN CONSIDERATIONS

The primary consideration in the design of MultiRPC was that it be inexpensive. We did not want normal RPC calls to be slowed down because of MultiRPC. Although one-to-many RPC calls constituted a very important special case, we expected simple, one-to-one RPC calls to be preponderant. A related, but distinct, concern was the increase in program size resulting from MultiRPC. Since virtual memory usage in our workstations was already high, we wished to keep MuhiRPC small.

Another influence on our design was the desire to decouple the design of subsystems from considerations relating to MultiRPC. We did not want to require any changes to clients who used only RPC2, or to servers. Our view was that only clients who wished to access multiple sites in parallel should have to know about MultiRPC.

Since we insisted on allowing simple RPC and MultiRPC calls in any order on any combination of connections, MultiRPC had to be completely orthogonal to normal RPC2 features. The delivery semantics, failure detection, support for multiple security levels, and other functions of RPC2 had to be retained when making a MultiRPC call.

A number of the scenarios in which we envisaged MultiRPC being used suggested that replies be processed by the client as they arrived rather than in batches. For example, in a quorum consensus replication scheme it is pointless to wait for messages beyond the required quorum. Since the exact nature of such processing is application dependent it has to be performed by a client-specified procedure. In addition, we felt that it was important for a client to be able to abort the MultiRPC call after examining any reply *or after* a specified amount of time had elapsed since the start of the MultiRPC call.

Finally, we wanted MuhiRPC to be simple to use. We have been successful in this even though the syntax of a MultiRPC call is different from the syntax of a simple RPC call. We have violated this syntax for two reasons: to allow clients to specify an arbitrary reply-handling procedure in a MultiRPC call, and to avoid the expansion in code size that would occur if individual MultiRPC stubs were generated.

IV. OVERVIEW OF RPC2

RPC2 consists of two relatively independent components: a Unix-based run-time library written in C, and a stub generator, *RP2Gen*. The run-time system is self-contained and is usable in the absence of *RP2Gen*. The code in the stubs generated by *RP2Gen* is, however, specific to RPC2.

A *subsystem* is a set of related remote procedures that make up a remote interface. *RP2Gen* takes a description of a subsystem and automatically generates code to marshal and unmarshal parameters in the client and server stubs. It thus performs a function similar to Lupine in the Xerox RPC mechanism [2], Matchmaker in Accent IPC [10], and HRPC from the University of Washington [15].

The RPC2 run-time system is fully integrated with a lightweight process mechanism (LWP) [14] that supports multiple nonpreemptive threads of control within a single Unix process. When a remote procedure is invoked, the calling LWP is suspended until the call is complete. Other LWP's in the same Unix process are, however, still **runnable**. The LWP package allows independent threads of control to share virtual memory, a feature that is not present in 4.2BSD Unix. Both RPC2 and the LWP package are entirely outside the Unix kernel and have been ported to multiple machine types. The **only** functionality required of the low-level packet transport mechanism is the ability to send and receive datagrams. At present, RPC2 runs on the DARPA IP/UDP protocol [6],[7].

RPC2 provides logical *connections*, whose relevant properties are as follows.

- 1) A connection is created when a client invokes the `BIND` primitive* and is destroyed by the `UNBIND` primitive.
- 2) Connections use little storage. Typically a connection requires a hundred bytes at each of the client and server ends. No other resources are used by a connection.
- 3) Within each Unix process, a connection is identified by

¹ Throughout this paper we use the term "packet" to mean a logical packet. In some networks, a packet may be physically transmitted as multiple fragments. Such fragmentation is transparent to RPC2.

² The cost of a `BIND` is comparable to the cost of a normal RPC.

a unique *handle*. Handles are never reused during the life of a process.

4) At any given time, a Unix process can have at most 64K active connections. This is about two orders of magnitude larger than the number of **connections** in use in the most heavily loaded Andrew servers.

Although one speaks of “clients” and “servers,” it should be noted that the mechanism is completely symmetric. A server can be a client to many other servers, and a client may be the server to many other clients. Although the roles of the peers are fixed on a given connection, a pair of hosts can act as client and server to each other by using two separate connections.

Tables I-III summarize the RPC2 primitives relevant to this paper. Besides `BIND` and `UNBIND`, the most important **runtime** primitive on the client side is `MAKERPC`. This call sends a request packet to a server and then waits for a reply. Reliable delivery is guaranteed by a retransmission protocol built on top of the **datagram** transport mechanism. Calls may take any arbitrary length of time; in response to client retries, the server sends **keep-alives** (`BUSY` packets) to indicate that it is still alive and connected to the network. On the server side, the basic primitives are `GETREQUEST`, which blocks until a request is received, the `SENDRESPONSE`, which sends out a reply packet. RPC2 provides *exactly-once* semantics in the absence of site and hard network failures, and *at-most-once* semantics otherwise [18].

A unique aspect of RPC2 is its support of arbitrary *side effects* on RPC calls. The side effect mechanism allows application-specific protocols such as file transfer to be efficiently integrated with the base code. The RPC2 design document discusses side effects in detail. For the purposes of this paper, it need only be noted that MultiRPC preserves side effect semantics.

V. MultiRPC AS AN EXTENSION TO RPC2

In this section, we first present the design of MultiRPC. We discuss certain reliability and performance problems revealed by a prototype implementation and then describe the **refinements** made to alleviate these deficiencies.

A. Overall Structure

Run-time support for MultiRPC is provided by the routine `MULTIRPC` that takes a request packet, a list of connections, and a *client handler* routine as input, and blocks until all responses have been received or until the call is explicitly terminated by the client handler. The packet which is transmitted to a server is identical to a packet generated by an RPC2 call. A server is not even *aware* that it is participating in a MultiRPC call. MultiRPC provides the same correctness guarantees as RPC2, except when the client terminates a call prematurely. In this case, a success return code indicates that no connection failures were detected prior to the point of termination. However, undetected server failures may have occurred after termination.

Language support for MultiRPC is provided by a pair of parameterized stub routines. These routines, `MAKEMULTI` and `UNPACKMULTI`, interpret templates called *argument descrip-*

tor structures (**ARG**'s) generated by `RP2Gen` to perform the packing and unpacking of parameters in request and reply packets. The decision to interpret **ARG**'s at run time rather than to use precompiled stubs as in RPC2 was motivated by storage size considerations. The slight additional processing cost of parameter interpretation is outweighed by the savings in the code size. A consequence of this is that the syntax of a MultiRPC call no longer resembles invocation of a local procedure. Each component of a MultiRPC call does, however, retain the semantics of an equivalent RPC2 call. The MultiRPC primitives are summarized in Table IV.

Appendix II describes the external interface of MultiRPC using an example. It presents a simple RPC2 subsystem in Fig. 1 and shows typical client and server code written by a user for non-MultiRPC calls in Figs. 3 and 4. `RP2Gen` uses the subsystem definition to generate a header file (Fig. 2) as well as client and server stub files (not shown). Fig. 5 shows how the user has to **modify** the client code to use MultiRPC. There are only two significant changes: the direct call to the client stub is replaced by an indirect call `viMAKEMULTI`, and an optional client handler routine is provided to process replies.

The **ARG**'s used by `MAKEMULTI` are declared by `RP2Gen` in the client stub file and pointers to them are defined in the header file. Each routine in a subsystem has an associated array of **ARG**'s, with the type, usage, and size of each parameter being specified by one array element. Structures are described by an array of **ARG**'s, one ARG per field. Nested structures are described by correspondingly nested **ARG**'s. At run time, `MAKEMULTI` traverses the ARG array and actual parameter list in step.

The client handler is activated exactly once for each connection specified in the MultiRPC call. Each activation corresponds to the receipt of a reply or to detection of a failure on that connection. The handler enables these events to be processed as soon as they occur. Its return code indicates whether the MultiRPC call should be continued or terminated.

The internal routine `SENDPACKETSRELIABLY` is the heart of the MultiRPC retransmission, failure detection, and result gathering mechanism. It performs an initial transmission of requests on all relevant connections and then awaits replies or timeouts. Each timeout on a connection causes a retransmission of the request to the corresponding server. On a reply, appropriate side effect processing is performed and then `UNPACKMULTI` is invoked. Client-specified timeouts are handled in this routine.

B. Handling Failures

Two factors complicate the semantics of failures in MultiRPC. First, since multiple connections are involved, how does one treat failures on an individual connection? Should the entire call be declared a failure and aborted at that point? In our design, this decision is delegated to the client handler. This routine is called on each failure and the return code from it specifies whether the call should be terminated. This allows applications to use a variety of strategies, such as termination on a single failure or termination beyond a threshold of failures.

The second source of complexity arises from the fact that the client handler can terminate a MultiRPC call before all

replies are received. What is the state of the connections on which replies have not been received? Should these connections be monitored for failure after the call is terminated? How are the outstanding replies dealt with if they do arrive eventually? Our strategy is to pretend that a response has actually been received on each of the outstanding connections. After termination of a call, MuhiRPC increments the sequence number and resets the state on each such connection. Responses that do eventually arrive are ignored. New failures will not be detected until the next MultiRPC or RPC2 call.

The ability to terminate a MultiRPC call prematurely interacted with an orthogonal aspect of RPC2 to produce a race condition. Originally, the RPC2 protocol required the client to send an acknowledgment to the server when a reply was received. The server would retry the reply until it received the acknowledgment or until it timed out. Suppose a client were to terminate a MultiRPC call prematurely and then immediately make another MultiRPC call. Then deadlock could arise on each connection on which a reply was outstanding when the first call was terminated. The retried replies by the server on that connection would be ignored by the client. Similarly, the server would ignore the new request from the client. Only a server or client timeout **could** end the deadlock. This problem would be compounded if the client terminated the second call prematurely, and then continued with further MultiRPC calls. The client could continue indefinitely in this mode without realizing that the connection was functionally dead.

Our original solution to this problem was to send an explicit negative acknowledgment if a packet with a sequence number higher than expected were received. This enabled both the client and the server to immediately detect the failure mode described above. We have since modified RPC2 to no longer require acknowledgments to replies. This obviates the problem mentioned above and solves another problem arising from an unexpected interaction with the underlying reliable transmission protocol. In the original protocol, the acknowledgment to a reply was often piggybacked on the next request. Unfortunately, this optimization did not work well in MultiRPC when many servers were involved. The total time to send out **all** the requests was large enough that the first servers to respond would timeout and retransmit their reply. This increased both the total number of packets exchanged as well as memory and processor utilization at the client. The additional processing further slowed the client and caused it to lose new replies, thus leading to an unstable mode of operation. The change to the protocol has successfully addressed both these problems. However, we still retain the negative acknowledgment mentioned earlier to allow prompt identification of connections that have been marked unusable by a server for other reasons such as side-effect failures.

Another possible failure mode relates to the client **handler** routine. During an excessively long computation in this routine, internal Unix buffers could be **filled** with incoming replies, causing further replies to be lost. This has the effect of increasing **retransmissions** and hence degrading performance. In addition, logical errors can arise if the client handler is not reentrant but yields control. This can happen, for instance, if the client handler makes an RPC during its processing. **Writ-**

ing a **client** handler is thus reminiscent of writing an interrupt handler for an operating system.

C. Evolution

Experience with an initial prototype of MultiRPC led us to make a number of changes pertaining to function and performance. The changes relating to function have been mentioned in Section V-B. In this section, we describe the changes that we made to improve the performance of MultiRPC.

Early trials of MultiRPC showed a surprisingly large number of retried packets, even when the number of servers being contacted was relatively small. Careful examination of the code showed that most of these packets were not being lost, but were being discarded after receipt. It turned out that the low-level RPC2 code first checked for timed-out events and then checked for packet arrivals. For a MultiRPC call to many sites, the total time to transmit all requests exceeded the first retransmission interval. Replies from the first few servers were discarded because they corresponded to events that had timed out. To fix this problem, we now time out events only after receiving all packets that have arrived.

Another change was made to the same piece of low-level code to reduce the number of LWP context switches. Rather than yield control on each received packet, the code now yields control only **after** all available packets have been received. In MuhiRPC, this reduces context switches because all these packets are destined for the same client LWP. This is in contrast to the situation in simple RPC2 calls where the semantics of RPC guarantees that a client LWP can be waiting for at most one packet.

A third change addressed the fact that Unix provides only a limited amount of buffering in the kernel for incoming packets. **For** a sufficiently large number of servers in a MultiRPC call, enough replies could arrive while requests were still being sent that the kernel buffer could overflow. To reduce the likelihood of this happening, we now yield control periodically when sending packets, allowing the low-level RPC2 code to receive replies and empty the kernel buffer.

Memory allocation is another area where we have made changes. A number of internal data structures are associated with each connection for the duration of a MultiRPC call. Although the number of connections in a MultiRPC call is known only at run time, our prototype statically allocated these data structures for simplicity. As is to be expected of static allocation schemes, this limited the maximum number of sites that could participate in a call and at the same time required a substantial amount of memory to be permanently allocated. We now allocate data structures as needed, but never deallocate them. This dynamic resizing avoids the limitations of static allocation yet avoids memory allocation on every **MultiRPC** call. Reallocation is necessary only when a MultiRPC call contacts more sites than have ever been contacted before by that process.

The extension to MultiRPC revealed a number of hidden assumptions in the underlying RPC2 code. In particular, there were at least two code segments whose processing time was quadratic in the number of outstanding events. This was never a problem of simple RPC2 calls since there could only be

one outstanding event per LWP, and since most applications used only a few LWP's. These pieces of code have now been modified to be linear, rather than quadratic.

The current implementation of MultiRPC incorporates all the modifications described in this section and a number of other minor changes. The performance measurements described in Section VI were obtained with this implementation.

VI. PERFORMANCE

The performance measure that best characterizes MultiRPC is the ratio of the elapsed time for using RPC2 iteratively (r), to the elapsed time for using MultiRPC (m). This ratio (r/m), as a function of the number of sites contacted (n), is the **speedup** realized by a MultiRPC implementation. Although linear **speedup** is clearly desirable, MultiRPC can be valuable even with modest **speedup**. In the application which motivated MultiRPC, for instance, rapid failure detection was of much greater concern than **speedup** of processing. This benefit of using MultiRPC exists even if there is no **speedup** of processing.

In this section, we assess the **speedup** of MultiRPC in three steps. We first present an analytic model in Section VI-A, validate this model using data from controlled experiments in Section VI-B2, and then present, in Section VI-B3 data from large-scale experiments where the assumptions behind our model are violated. The raw data for the measurements and the analytic model predictions are found in Appendix IV, and are presented graphically in Appendix V.

A. Analytic Model

Our goal in this section is to derive an analytic model that can predict the behavior of MultiRPC. Although the simplifying assumptions we make may not strictly hold in practice, they are acceptable for the level of accuracy we are trying to achieve.

The most important assumption deals with network topology and latency. In most distributed systems, the actual transit time on the network is a small fraction of the processing time spent in sending and receiving a packet. Routers or other interconnecting elements on a multisegment network can, however, increase latency considerably. For the purposes of our model, we assume that the client and all the servers are on a single-segment network that has negligible latency.

A second assumption relates to mutual interference and loss of packets. Although MultiRPC is built on unreliable datagrams, the actual probability of packet loss is quite low, typically below 1 percent. However, as more servers are contacted, the probability of packet loss increases because of limited buffering capability at the client. Even if packets are not lost, race conditions between the client and the servers can cause packet retransmissions. We ignore all these complications and assume that there are no lost or retried packets during a MultiRPC call.

Finally, we assume that each server takes a constant amount of time to service a request and that this time is uniform across all servers. This assumption is valid to a first approximation even though the specific nature of the request, the presence of other processing activity at the servers, and slight differences

in hardware performance can result in nonuniform service times.

A MultiRPC call can be decomposed into the following components:

- pack* Packing of arguments by client.
- cloh* Protocol and kernel processing by client to send request.
- servoh* Protocol and kernel processing by server to receive request and send reply.
- clproc* Protocol and kernel processing by client to receive reply.
- unpack* Unpacking of arguments and processing in client handler.

In terms of the MultiRPC implementation described in Section V-A, *pack* is the time taken by the routine MAKEMULTI, *cloh* corresponds to the time in MULTIRPC and the initial part of SENDPACKETSRELIABLY, *clproc* corresponds to the remainder of SENDPACKETSRELIABLY, and *unpack* is the time taken by UNPACKMULTI and a call to a null client handler routine. Since MultiRPC and simple RPC2 calls are indistinguishable at the servers, *servoh* is the same for both iterative RPC2 calls and MultiRPC. This is the total time taken to receive a request and to send a reply, assuming zero processing time. We include a separate term *comptime* to account for the application processing at a server.

The *pack* component is performed only once, regardless of the number of servers being contacted. The *servoh* and *comptime* components overlap at the servers. All the other components have to be performed once for each server. In terms of these components, the total time m for a MultiRPC call to n sites can be expressed as

$$m = pack + (n \times cloh) + (servoh + comptime) + (nb \times clproc) + (n \times unpack).$$

Unfortunately this expression contains an oversimplification that affects the model predictions significantly. Suppose *waittime* is the elapsed time between the sending of the last request and the receipt of the first reply by the client. For a single server, *waittime* will be the sum of *servoh* and *comptime*. For a large enough number of servers, however, the reply from the first server may be available before the last request is sent out: in this case, *waittime* is zero.

Assuming that the time to send a packet is *sendtime*, the expression for m can be refined as follows:

$$waittime = (servoh + comptime) - ((n - 1) \times sendtime)$$

$$\text{if } (waittime < 0) \text{ then } waittime = 0$$

$$m = pack + (n \times cloh) + waittime + (n \times clproc) + (n \times unpack).$$

If a null RPC2 call takes *rpctime*, the total time r to contact n servers using iterative RPC2 calls is given by

$$r = n \times (rpctime + comptime).$$

The times for the individual components in our implementation were obtained by actual measurement and are presented in Table VI. Using these values in the expressions derived above we can calculate the quantities r , m , and r/m for server computation times of 10, 20, and 50 ms. These predicted values are presented in Table VII and shown graphically in Fig. 8.

Our choices of server computation times are indicative of those in the Andrew file system. Many calls require one disk access, which takes about 20 ms. The 50 ms time is representative of a heavily loaded server or one servicing a more complex request. 10 ms is the smallest delay we were able to simulate reliably in our experiments.

Most systems with parallelism initially exhibit linear **speedup**, then show sublinear **speedup**, and finally saturate. Fig. 8 shows that MultiRPC conforms to this expected behavior. However, two detailed **observations** are apparent from this graph. First, saturation occurs at surprisingly low levels of **speedup**. Second, the level at which **speedup** saturates and the number of servers at which saturation sets in are both dependent on the server computation time.

The server computation time is central to MultiRPC because most of the parallelism comes from overlap of server computations. The sending of requests and processing of replies at the client are done sequentially. The realizable **speedup** depends on how long these operations take in comparison to the time spent at the server.

We can quantify this reasoning in the following way. Our measurements show that the dominant components of MultiRPC are the time to send a request and the time to receive a reply. Suppose the sum of these quantities, called *systemtime*, is identical in MultiRPC and RPC2. Furthermore, let the total time spent at a server (equal to the sum of *servoh* and *comptime*) be *servtime*. Then the MultiRPC and RPC2 call times to n servers can be crudely approximated as follows:

$$m = (n \times \text{systemtime}) + \text{servtime}$$

$$r = (n \times \text{systemtime}) + (n \times \text{servtime})$$

$$\frac{r}{m} = \frac{\text{systemtime} + \frac{\text{servtime}}{n}}{\text{systemtime} + \frac{\text{servtime}}{n}} = \frac{1 + \frac{\text{servtime}}{\text{systemtime}}}{1 + \frac{\text{servtime}}{\text{systemtime}} \times \frac{1}{n}}$$

Let

$$T = \frac{\text{servtime}}{\text{systemtime}},$$

Then

$$\frac{r}{m} = \frac{1 + T}{1 + \frac{T}{n}}$$

The above expression clearly shows that the **speedup** is sensitive to the value of T . In the **limit**, as n tends to infinity, the value of r/m tends to $1 + T$. This accounts for the fact that the saturation value of the **speedup** is higher for longer server computation times. Because the times to send and receive a packet dominate *systemtime*, improvements to the underlying network primitives will improve the maximum **speedup** obtained

with MultiRPC. Consequently, an improved basic RPC mechanism would result in improved MultiRPC performance rather than rendering it superfluous.

B. Experimental Results

We performed a series of carefully controlled experiments to confirm our understanding of MultiRPC and to explore its behavior when contacting a large number of servers. We describe our experiments and the observations from them in the next three sections.

1) *Experimental Environment*: The experiments were conducted in an environment of about 500 Sun3, DEC MicroVax, and IBM RT-PC workstations running the Unix 4.2BSD operating system and attached to the Andrew File System. For uniformity, we ran our tests only on the IBM RT-PC workstations, with one of the workstations being the client and the other servers. By designing our tests to require no file accesses or system calls on the servers we avoided distortions of our measurements by distributed file system access.

The topology of the network connecting these workstations is shown in Appendix III. Although physically compact, there is considerable complexity in the network structure. There are about a dozen Ethernet and IBM Token Ring **subnets** connected to each other directly or via optic fiber links. Active computing elements, called **routers**, perform the appropriate forwarding or filtering of packets between these **subnets**. In addition to the Andrew **workstations**, many standalone workstations and mainframe computers are also on this network.

The scale and complexity of the network introduced serious problems in controlling our experiments. We had to be on guard against extraneous network activity loading the network and the routers. We also had to contend with the fact that closely-spaced replies to a MultiRPC call from a large number of servers could overload the routers and affect our measurements.

To address these problems we separated our experiments into two classes. The tests for model validation, discussed in Section VI-B2, were run entirely on workstations located on a single **subnet**. For these tests, we were able to ensure that there was no other network activity. The presence of routers was irrelevant since the client and all the servers were on the same **subnet**. Since there were only 20 workstations on this **subnet**, our model validation is restricted to this range.

The tests discussed in Section VI-B3 to explore the large-scale performance of MultiRPC could not be controlled so well. To include more than 20 servers our tests had to span multiple **subnets**. We minimized the effects of extraneous network activity by running our experiments in the early hours of the morning, when the network was least active. Preliminary tests confirmed that this consistently produced smaller variances in our measurements than tests run at any other period of the day.

We simulated computation on the servers by delaying the reply to a request by a specified amount of time. Unfortunately, the clock resolution of 16 ms on the RT-PC's was inadequate for the range of computation times of interest to us. We there-

³ The entire CMU campus is only about one square mile in size.

fore had to resort to a timing loop to achieve delays of 10, 20, and 50 ms in our tests. The clock resolution was also inadequate for measuring the elapsed time of individual calls at the client. To overcome this, we timed many iterations of each call and used the average.

2) *Validation*: The worst performance of MultiRPC, relative to RPC2, occurs when a single site is contacted. In this situation, the additional complexity of a MultiRPC call is not amortized over many connections. Table V compares the elapsed times to contact a single site using RPC2 and MultiRPC, for zero server computation time. The table shows that the difference in times is negligible. Furthermore, the RPC2 time presented in the table is not worse than the time observed on an earlier version of RPC2 that lacked MultiRPC support. Our design criterion of not slowing down simple RPC's has thus been met.

To compute the model predictions we need to know the times taken by individual components of MultiRPC. These values, obtained by standalone measurements of MultiRPC, are presented in Table VI. By substituting these values in the expressions derived in Section VI-A, we obtain the model predictions shown in Table VII and Fig. 8.

Figs. 9-11 compare the predictions of the model to our measurements. The model consistently predicts slightly better performance than we actually observe, but the fit is surprisingly good considering the simplicity of the model. For the reasons discussed earlier we were unable to investigate the validity of the model beyond 20 servers.

3) *Large Scale Effects*: Since the original motivation for MultiRPC involved a scenario with a large number of workstations distributed over the entire CMU campus, we were curious to see just how well MultiRPC behaves in such an environment. Our analytical model is not valid in this situation because of the presence of routers and extraneous network traffic. These factors affect both RPC2 and MultiRPC. Their effects show up as anomalies in the average values of the measured quantities, and as high associated variances.

Table VIII and Fig. 12 present our observations for server computation times of 10, 20, and 50 ms. The behavior below about 25 servers is in accordance with our model. Beyond this, the **speedup** drops rather than increasing or remaining constant. Detailed examination of the data revealed an increase in the number of retried packets beyond about 25 servers. Below 25 servers, retries never comprised more than 0.4 percent of the total packets sent in a MultiRPC test. In fact, for those configurations there were often no retried packets at all. Beyond 25 servers the number of retried packets increased, and sometimes accounted for as much as 25 percent of the total traffic in configurations beyond 50 servers.

The measurements described above were performed with server computation times that were constants. We conjectured that one of the main reasons for poor behavior at large scale was the overloading of routers due to simultaneous arrival of replies from many servers. Real server computations tend not to be constant. We therefore repeated the experiments with computation times normally distributed, with a standard deviation that was 10 percent of the mean. Table IX and Fig.

13 show the corresponding results. We also repeated the experiments with an exponentially distributed service time, to provide validation data for a stochastic model that might be built in the future. These data are shown in Table X and Fig. 14. The aberrations in performance at large scale do not, however, vanish when using a nonconstant service time distribution. In all cases, large standard deviations make it difficult to interpret the data for 100 servers.

Although there is a decline in observed **speedup** in the large scale tests, it must be emphasized that **MultiRPC** performance is always better than iterative RPC2 performance. Furthermore, we encountered no functional problems in using MultiRPC up to 100 servers. These facts give us confidence in the value of MultiRPC as a basic component of large distributed systems.

VII. RELATED WORK

In this section, we look at a number of parallel RPC mechanisms with a view to placing the design of MultiRPC in perspective. We make no attempt to be exhaustive in our examples or to be complete in our descriptions. Rather, our goal is to examine these systems in a manner that highlights their similarities and differences with respect to MultiRPC.

Work in the area of parallel network access has typically focused on broadcast or multicast protocols. Two examples of such work are the Sun Microsystems' Broadcast RPC (**Sun-BRPC**) [19], and Group Interprocess Communication in the V Kernel (**V-GIPC**) [4].

Sun-BRPC depends upon IP-level broadcast to communicate with multiple sites. Servers must register themselves in advance with a central port in order to be accessible via the broadcast facility. This is in contrast to MultiRPC where no explicit actions need be taken by the server; existing servers do not have to be modified, recompiled, nor relinked. Like MultiRPC, Sun-BRPC provides for the client handler routine and an overall client-specified timeout. However, it does not provide the same correctness guarantees and error reporting as MultiRPC.

V-GIPC uses the Ethernet multicast protocol as its basis and defines *host groups* as message addresses. Each request is multicast and it is up to each host to recognize those group addresses for which it has local processes as members. Reliable communication is not an objective of V-GIPC, even though its designers report that lost responses due to simultaneous arrival of packets are common. Another difference is that there is no notion of a client handler in V-GIPC. A client is blocked only until the first response is received; further responses have to be explicitly gathered. When another call is made, the previous call is implicitly terminated and further responses to it are discarded.

Neither Sun-BRPC nor V-GIPC allows long server computations at all sites in a parallel call while providing timely notification of site or network failures. A sufficiently long computation would simply cause a timeout. The **MultiRPC** retransmission protocol addresses this issue and allows the client to distinguish between a long computation and permanent communication failure.

There are also parallel RPC systems which do not depend on broadcast nor multicast. One such system is Circus [5], which focuses on achieving high availability by using parallel RPC as a vehicle for replication. Circus is built on top of the DARPA IP/UDP protocol and is unique in that it supports many-to-many communication rather than one-to-many. It provides for a fixed set of routines called *collators*, one of which must be specified by the client when making a call. Collators perform a function similar to the MultiRPC client handler routine, but differ in that they do not allow a call to be terminated prematurely. Like MultiRPC, Circus uses probe packets to distinguish between long server computations and permanent communication failure. Many problems addressed by Circus, such as orphan detection and exactly-once semantics in the presence of failures, are unique to its intended application.

The Gemini parallel RPC mechanism [3],[12] is built on the reliable IP/TCP byte stream protocol [8], that subsumes the retransmission, timeout, acknowledgment, and probe functions of RPC. It is similar to MultiRPC in that it requires no multicast or broadcast support. Unlike MultiRPC, a distinct Unix process is created on a server for each client. The stub compiler for Gemini accepts interface specification in C rather than defining a separate interface language. The equivalent of the MultiRPC client handler routine is a language construct called a *result statement*. This is a compound statement in C that syntactically appears after a parallel remote procedure call. This body of code is executed exactly once for each reply and the execution can terminate the entire call prematurely.

Stream-Calls in Mercury [11] can be used to provide a function similar to MultiRPC. The ability to make multiple requests and to defer receipt of replies allows a client to have multiple calls outstanding to one or more sites. In contrast to MultiRPC which retains the strictly synchronous character of RPC, this use of stream-calls is closer to a message-passing paradigm.

VIII. CONCLUSION

The central message of this paper is that it is possible to build an efficient and easy to use parallel invocation mechanism whose semantics is a natural extension of the remote procedure call paradigm. We have derived an analytic model of this mechanism and shown that its predicted performance closely matches the measured performance of our implementation. Asymptotic analysis of this model indicates that improvements to the underlying transmission primitives would further strengthen the case for using MultiRPC in preference to iterative RPC's. We have demonstrated experimentally that the

mechanism works successfully for up to 100 servers in a single call, executing in a complex network environment with diverse transmission media and interconnecting elements. Comparison to other parallel invocation mechanisms shows that MultiRPC is unique in its overall design, although some of the individual concepts used in it may be found elsewhere.

Although the experiments presented in this paper were performed on a high-speed local area network, we are convinced that MultiRPC will be of equal or greater value in the context of a slower, wide-area network. Such a network will require longer timeout and retransmission intervals from the underlying remote procedure call mechanism. Consequently the rationale for MultiRPC presented in Section II will apply even more strongly.

There are a number of ways in which the work reported here may be extended. First, MultiRPC provides parallelism at the programming interface but does not require multicast capability in any of the lower levels of the networking software. Suppose, however, multicast were available. How could MultiRPC use it? What would its performance behavior be then? Our view is that multicast is a performance enhancement rather than a fundamental programming primitive. Preliminary work indicates that if the programmer is required to explicitly group connections, the semantics of MultiRPC can be preserved while using multicast internally. An interesting security question arises in this context. How does one support site-specific encryption on multicast packets? Our approach is to use the underlying secure RPC2 connections as key distribution channels for internally generated group-specific keys. We will report on our experience with this approach and further details on the use of **multicast** in a later paper.

Another potential improvement of MultiRPC involves **ARG's**. **RP2Gen** does not take advantage of repeated argument types when it generates **ARG's**; it creates a new ARG for each parameter of each remote operation. For recursive structure arguments this can consume a significant amount of storage. Since structure arguments are defined in the subsystem specification tile, and hence known to **RP2Gen**, it should be possible to share **ARG's**.

Finally, using software in a variety of applications often results in unexpected lessons and refinements. It is impossible to predict in advance what such changes might be in the context of MultiRPC. However, we are **confident** that MultiRPC and mechanisms similar to it will prove to be an important building block in distributed systems.

APPENDIX I

SUMMARY OF PRIMITIVES

TABLE I
RPC2 CLIENT PRIMITIVES

Primitive	Description
BIND	Create a new connection
MAKERPC	Make a remote procedure call

TABLE II
RPC2 SERVER PRIMITIVES

Primitive	Description
EXPORT	Indicate willingness to accept calls for a subsystem
DEEXPORT	Stop accepting new connections for one or all subsystems
GETREQUEST	Wait for an RPC request or a new connection
ENABLE	Allow servicing of requests on a new connection
SENDRESPONSE	Respond to a request from a client
INITSIDEEFFECT	Initiate side effect
CHECKSIDEFFECT	Check progress of side effect

TABLE III
MISCELLANEOUS RPC2 PRIMITIVES

Primitive	Description
INIT	Perform runtime system initialization
UNBIND	Terminate a connection by client or server
ALLOCBUFFER	Allocate a packet buffer
FREEBUFFER	Free a packet buffer

TABLE IV
MultiRPC PRIMITIVES

Primitive	Description
MULTIRPC	Initiate a parallel remote procedure call
MAKEMULTI	Perform parameter marshalling for MultiRPC
UNPACKMULTI	Perform parameter unmarshalling for MultiRPC

APPENDIX II AN EXAMPLE

This Appendix presents a brief example in order to make some of the previous discussion more concrete. Although this example is contrived, it is adequate to illustrate the structure of a client and server that communicate via RPC2, and the changes that must be made to the client to use MultiRPC.

The subsystem designer defines a subsystem, chooses a name for it, and writes the specifications in the file *<subsystemname>.rpc2*. This file is submitted to RP2Gen, which generates client and server stubs and a header file. RP2Gen names its generated files using the file name of the *.rpc2* file with the appropriate suffix.

Fig. 1 presents the specification of the subsystem *example* in the file *example.rpc2*. RP2Gen interprets the specification and produces a client stub in the file *example.client.c*, a server stub in *example.server.c*, and a header file *example.h* (shown in Fig. 2). This subsystem is composed of two operations, *double-it* and *triple-it*. These procedures both take a call-by-value-result parameter, *testval*, containing both the integer to be operated on and the result returned by the server.

Once the interface has been specified, the subsystem imple-

mentor is responsible for exporting the subsystem and writing the server main loop and the bodies of the procedures to perform the server operations. A client wishing to use this server must first bind to it and then perform an RPC on that connection. Figs. 3 and 4 illustrate the client and server code. *We wish to emphasize that this code is devoid of any considerations relating to MultiRPC.*

Now consider extending this example to contact multiple servers using MultiRPC. The *example.rpc2* file and the server code remain exactly the same. Argument descriptor structures (ARG's), used by MultiRPC to marshal and unmarshal arguments, are already present in the client stub file; pointers to these structures are defined in the *example.h* file. Only the client code has to be modified, as shown in Figs. 5 and 6.

From the client's perspective, a MultiRPC call is slightly different from a simple RPC2 call. The procedure invocation no longer has the syntax of a local procedure call. Instead, the single library routine `MAKEMULTI` is used to access the run-time system routine `MULTIRPC`. The client must allocate all necessary parameter storage. IN arguments are simply supplied as for any procedure call, but for OUT and IN_OUT parameters arguments arrays of pointers to the appropriate types must be supplied to the `MAKEMULTI` routine. The return arguments from

RP2Gen specification file for simple subsystem

```

Server Prefix "serv";
Subsystem "Example";

#define EXPORTAL 3000
#define EXSUBSYS 151

```

tag sewer operations to avoid ambiguity

*logical portal and subsystem numbers
chosen by subsystem designer*

Procedure specifications

```

double__it(IN OUT Integer testval);
triple__it(IN OUT Integer testval);

```

Fig. 1. *example.rpc2* specification file*.h file produced by RP2GEN**Input file: ex.rpc*

```

#include "rpc2.h"
#include "se.h"
#define EXPORTAL 3000
#define EXSUBSYS 151

```

*logical portal and subsystem numbers
defined in example.rpc2*

Op codes and definitions

```

extern long double__it();
#define double__it__OP 1
extern ARG double__it__ARGS[];
#define double__it__PTR double__it__ARGS

extern long triple__it();
#define triple__it__OP 2
extern ARG triple__it__ARGS[];
#define triple__it__PTR triple__it__ARGS

```

Fig. 2. The RP2Gen-generated header file *example.h*.

Include relevant header files

```

main()
{
  int testval, op;
  Handle cid;
  int rc, mylpid;
  HostIdent hident;
  PortalIdent pident;
  SubsysIdent sident;

  Perform LWP and RPC2 initialization
  InitializeProcessSupport(NORMAL_PRIORITY, &mylpid);
  Init(VERSION, 0, NULL, 1, -1, NULL);

  Initialize hident, pident and sident here

  Establish connection to server using Bind
  rc = Bind(OPENKIMONO, NULL, &hident, &pident, &sident, 0, NULL, NULL, &cid);
  if (rc != SUCCESS) printf("Bind: %s\n", ErrorMsg(rc));

  while (1)
  {
    printf("\nDouble [= 1] or Triple [= 2] (type 0 to quit)? ");
    scanf("%d", &op);
    if (op == 0) break;
    printf("Number? ");
    scanf("%d", &testval);

    Perform RPC2 call on connection cid
    switch(op)
    {
      case 1:
        rc = double__it(cid, &testval);      Invoke server
        break;
      case 2:
        rc = triple__it(cid, &testval);     Inwke stnw
        break;
      default:
        printf("Bad choice: %d\n", op);
        continue;
    }
    if (rc != SUCCESS) printf("Double/Triple: %s\n", ErrorMsg(rc));
    else printf("result = %d\n", testval);
  }

  Unbind(cid);      Terminate connection with server
  printf("Bye...\n");
}

```

Fig. 3. A simple RPC2 client

Include relevant header files

```

main()
{
  RequestFilter reqfilter;
  PacketBuffer *reqbuffer;
  int mylpid;
  Handle cid;
  PortalIdent portalid, *portallist[1];
  SubsysIdent subsysid;

```

Fig. 4. A simple RPC2 server.

```

Perform L WP and RPC2 initialization
InitializeProcessSupport(NORMAL_PRIORITY, &mylpid);

Specify portal on which server will belistening
portalid.Value.InetPortNumber = htons(EXPORTAL);
portallist[0] = &portalid;
init(VERSION, 0, portallist, 1, -1, NULL);

Export subsystem
subsysid.Tag = SUBSYSBYID;
subsysid.Value.SubsysId = EXSUBSYS;
Export(&subsysid);

Set filter to accept requests for the exported subsystem
reqfilter.ConnOrSubsys.SubsysId = EXSUBSYS;

Enter server loop
while(1)
{
    Await a client request:
    GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL);
    Example_ExecuteRequest(cid, reqbuffer); Unpacking routine generated by RP2Gen
}
}

Bodies of server procedures
long serv_double_it(cid, testval)
    Handle cid;
    Integer | tastval;
    {
        | terJval = (*testval) | 2;
        return(SUCCESS);
    }

long serv_triple_it(cid, testval)
    Handk cid;
    Integer *testval;
    {
        *testval = (*testval) * 3;
        return(SUCCESS);
    }

```

Fig. 4. (Continued).

```

Include relevant header files here

#define HOWMANY 3 howmany servers to calf
#define WAITFOR 2 wait for 2 out of 3 replies
Declare all handler routines here

long returns; reply counter

main()
{
    int | testval[HOWMANY], op; Can use static or dynamic allocation
    Handle cid[HOWMANY];
    HostIdent hident;
    PortalIdent pident;
    SubsysIdent sident;
    int count;

```

Fig. 5. Client using MultiRPC.

```

Perform LWP and RPC2 initialization
InitializeProcessSupport(NORMAL_PRIORITY, &mylpid);
Ini(VERSION, 0, NULL, 1, -1, NULL);

Initialize pident and sident here

for (count = 0; count < HOWMANY; count++)
{ Establish connections to servers:
  Initialize hident here
  Bind(OPENKIMONO, NULL, &hident, &pident, &dent, 0, NULL, NULL, &cid[count]);
  testval[count] = (int) malloc(sizeof(int)); allocate space for arguments
}

while (1)
{
  printf("\nDouble [ = 1] or Triple [ = 2] (type 0 to quit)? ");
  scanf("%d", &op); if (op == 0) break;
  printf("\nNumber? ");
  scanf("%d", testval[0]); IN argument goes in 1st array slot

  Make the MultiRPC call
  returns = 0;
  switch(op) {
    case 1: MRPC_MakeMulti(double_it_OP, double_it_PTR, HOWMANY, cid,
      HandleDouble, NULL, testval); Invoke servers
    break;

    case 2: MRPC_MakeMulti(triple_it_OP, triple_it_PTR, HOWMANY, cid,
      HandleTriple, NULL, testval); Invoke servers
    break;
    default: continue;
  }
}
for (count = 0; count < HOWMANY; count++)
  Unbind(cid[count]); Terminate connection with server
printf("Bye...\n");
}

```

Fig. 5. (Continued)

Client handler routines

```

long HandleDouble(HowMany, cidarray, host, rpcval, testval)
  Integer HowMany, host, rpcval, *testval[];
  Handle cidarray[];
  {
  if (rpcval != SUCCESS)
    printf("HandleDouble: rpcval = %s\n", ErrorMsg(rpcval));
  else printf("testval[%d] = %d\n", host, testval[host]);
  if (++ returns == WAITFOR) return -1; Terminate the MultiRPC call
  return(0); Continue accepting server responses
  }

long HandleTriple(HowMany, cidarray, host, rpcval, testval)
  Integer HowMany, host, rpcval, *testval[];
  Handle cidarray[];
  {
  if (rpcval != SUCCESS)
    printf("HandleTriple: rpcval = %s\n", ErrorMsg(rpcval));
  else printf("testval[%d] = %d\n", host, *testval[host]);
  if (++ returns == WAITFOR) return -1; Terminate the MultiRPC call
  return(0); Continue accepting sewer responses
  }

```

Fig. 6. MultiRPC client handler routines.

each of the servers will be placed in the appropriate elements of the arrays.

The client is also responsible for supplying a handler routine for any server operation which is used in a MultiRPC call. The handler routine is called by MultiRPC as each individual server response arrives, providing an opportunity to perform

incremental bookkeeping and analysis. The return code from the handler gives the client control over the continuation or termination of the MultiRPC call.

APPENDIX III
NETWORK TOPOLOGY

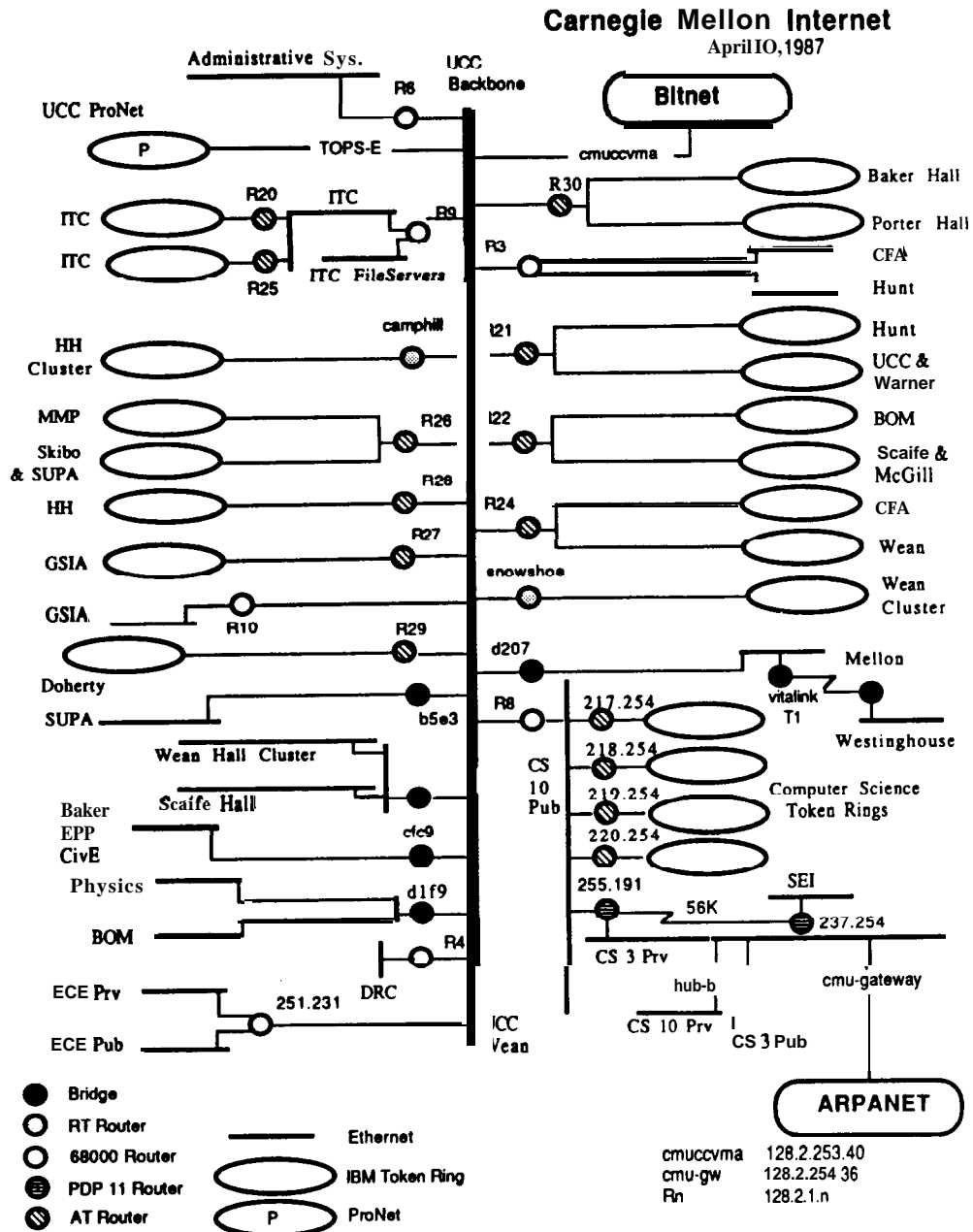


Fig. 7. Carnegie Mellon network topology.

APPENDIX IV

TABLES

TABLE V

MEASURED TIMES FOR NULL RPC2 AND MultiRPC CALLS
(IN MILLISECONDS)

MRPC	RPC2	R / M
29.00 (0.97)	27.80 (1.01)	0.96

TABLE VI
AVERAGE TIMES FOR INDIVIDUAL COMPONENTS OF MultiRPC (IN MILLISECOND_S)

<i>pack</i>	<i>cloh</i>	<i>servoh</i>	<i>clproc</i>	<i>unpack</i>	<i>sendtime</i>
0.56	5.98	13.69	3.93	0.20	5.15

TABLE VII
PREDICTED PERFORMANCE FROM ANALYTIC MODEL

servers	10 ms Computation			20 ms Computation			50ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	34.36	37.80	1.10	44.36	47.80	1.08	14.36	77.80	1.05
2	39.32	75.60	1.92	49.32	95.60	1.94	19.32	155.60	1.96
5	54.20	189.00	3.49	64.20	239.00	3.72	94.20	389.00	4.13
7	71.33	264.60	3.71	74.12	334.60	4.51	104.12	544.60	5.23
10	101.66	378.00	3.12	101.66	478.00	4.70	119.00	778.00	6.54
13	131.99	491.40	3.72	131.99	621.40	4.71	133.88	1011.40	7.55
15	152.21	567.00	3.73	152.21	717.00	4.11	152.21	1167.00	7.67
17	172.43	642.60	3.73	172.43	812.60	4.71	172.43	1322.60	7.67
19	192.65	718.20	3.73	192.65	908.20	4.71	192.65	1478.20	7.67
20	202.76	756.00	3.73	202.76	956.00	4.71	202.76	1556.00	7.67
25	253.31	945.00	3.73	253.31	1195.00	4.72	253.31	1945.00	7.68
50	506.06	1890.00	3.73	506.06	2390.00	4.72	506.06	3890.00	7.69
75	758.81	2835.00	3.74	758.81	3585.00	4.72	758.81	5835.00	7.69
100	1011.56	3780.00	3.74	1011.56	4780.00	4.73	1011.56	7780.00	7.69

TABLE VIII
MEASURED CALL TIMES FOR CONSTANT SERVER COMPUTATION TIMES

Servers	10 ms Computation			20ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MR PC	RPC2	R/M
1	39.30 (0.73)	38.30 (1.13)	0.97	49.70 (1.08)	47.30 (1.34)	0.95	81.10 (6.37)	78.80 (6.18)	0.97
2	45.90 (0.85)	76.50 (1.10)	1.67	56.40 (0.94)	96.10 (1.21)	1.70	85.M (7.70)	158.35 (7.43)	1.85
5	66.25 (1.07)	191.55 (2.19)	2.89	76.45 (0.94)	242.40 (4.08)	3.17	104.35 (7.63)	351.20 (3.52)	3.75
7	79.80 (1.15)	268.70 (3.77)	3.37	89.90 (1.02)	336.25 (3.35)	3.74	120.95 (8.68)	549.75 (6.56)	4.55
10	107.90 (1.21)	385.10 (11.50)	3.57	110.55 (1.10)	482.05 (7.78)	4.36	139.50 (790)	784.80 (9.57)	5.63
13	140.10 (1.07)	500.80 (9.51)	3.57	139.95 (1.00)	627.35 (7.74)	4.48	161.40 (7.24)	1046.30 (37.60)	6.48
15	161.30 (1.08)	579.10 (12.82)	3.59	161.35 (1.23)	725.95 (8.20)	4.50	175.20 (9.00)	1179.25 (14.02)	6.73
17	182.40 (0.99)	654.10 (10.47)	3.59	183.35 (1.50)	820.05 (8.31)	4.47	188.75 (10.07)	1337.50 (12.52)	7.09
19	204.10 (1.07)	740.85 (22.37)	3.63	205.60 (2.52)	915.55 (55.51)	4.45	209.35 (10.62)	1496.95 (17.00)	7.15
50	766.10 (26.30)	2744.00 (73.36)	3.58	766.50 (20.28)	3130.90 (59.34)	4.08	796.90 (26.21)	4912.50 (255.90)	6.16
75	'	'	*	1225.20 (114.13)	4985.53 (86.52)	4.07	1331.20 (433.06)	7707.90 (245.59)	5.79
100	1705.90 (27.00)	6017.20 (222.09)	3.53	1780.40 (136.46)	6844.00 (246.76)	3.84	1429.50 (217.27)	9947.00 (212.63)	6.96

All times are in milliseconds. Figures in parentheses are standard deviations. A subset of these data is graphically presented in Fig. 12 and is also used in Figs. 9-11. For configurations up to 20 servers, all machines were located on a single isolated token ring. Configurations involving more than 20 servers spanned multiple network segments. Each data point was obtained from ten trials.

TABLE IX
MEASURED CALL TIMES WITH NORMALLY DISTRIBUTED COMPUTATION TIMES

Servers	10 ms Computation			20 ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	39.32 (1.11)	37.75 (0.95)	1.02	50.40 (6.42)	47.60 (1.00)	0.94	77.96 (7.92)	78.40 (12.28)	1.01
2	44.92 (1.29)	75.92 (1.58)	1.69	59.56 (6.68)	99.88 (10.41)	1.68	85.28 (7.74)	155.96 (12.06)	1.83
5	70.16 (3.25)	208.52 (3.74)	2.97	79.32 (2.98)	261.68 (5.75)	3.30	108.36 (12.48)	414.24 (31.02)	3.82
10	112.55 (3.97)	478.10 (16.40)	4.25	112.08 (9.15)	530.00 (17.95)	4.73	143.40 (10.01)	824.76 (19.72)	5.75
15	162.75 (18.40)	747.75 (21.09)	4.59	148.16 (2.19)	783.36 (20.76)	5.29	185.55 (14.50)	1270.85 (30.83)	6.85
19	*	*	*	*	*	*	224.00 (25.12)	1698.70 (42.18)	7.58
20	208.05 (7.10)	956.55 (35.47)	4.60	197.44 (2.16)	1065.76 (21.33)	5.40	*	*	*
25	247.48 (1.19)	1153.52 (18.49)	4.66	248.96 (5.14)	1430.00 (93.75)	5.74	258.04 (13.50)	2179.00 (47.62)	8.44
50	780.00 (34.57)	2933.10 (124.90)	3.76	771.79 (8.56)	3266.20 (56.84)	4.23	838.20 (34.70)	4901.70 (186.16)	5.85
75	1474.30 (103.62)	4579.60 (57.35)	3.12	1238.20 (177.80)	5351.07 (239.53)	4.32	1581.50 (429.48)	7690.20 (277.17)	4.86
100	1715.89 (28.35)	6120.10 (61.95)	3.57	1637.30 (121.99)	7282.70 (403.44)	4.45	1943.60 (1965.90)	10082.40 (248.46)	5.19

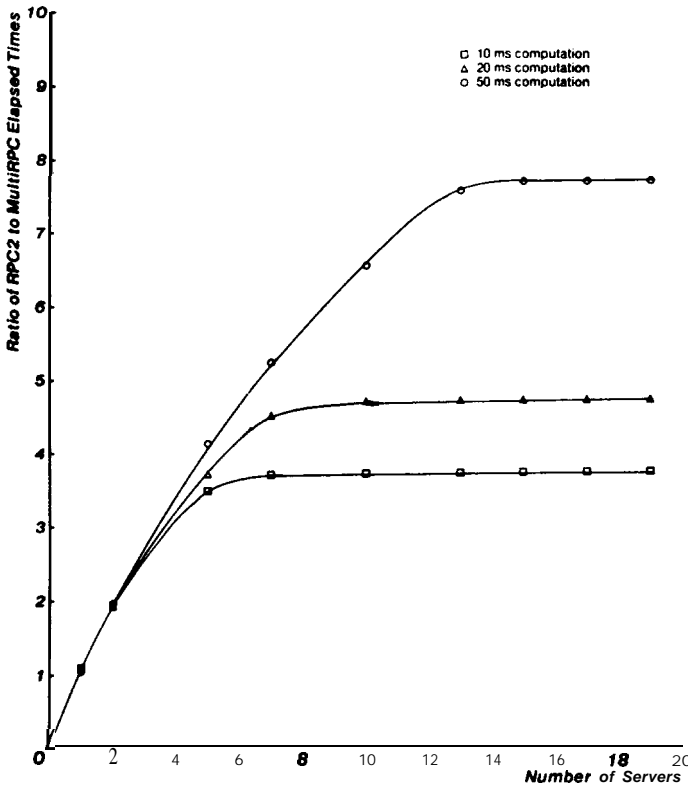
All times are in milliseconds. **Figures** in parentheses are standard deviations. These data were obtained from servers distributed over many network segments. A subset of these data is graphically presented in Fig. 13. Each data point was obtained from ten trials.

TABLE X
MEASURED CALL TIMES WITH EXPONENTIALLY DISTRIBUTED SERVER COMPUTATION TIMES

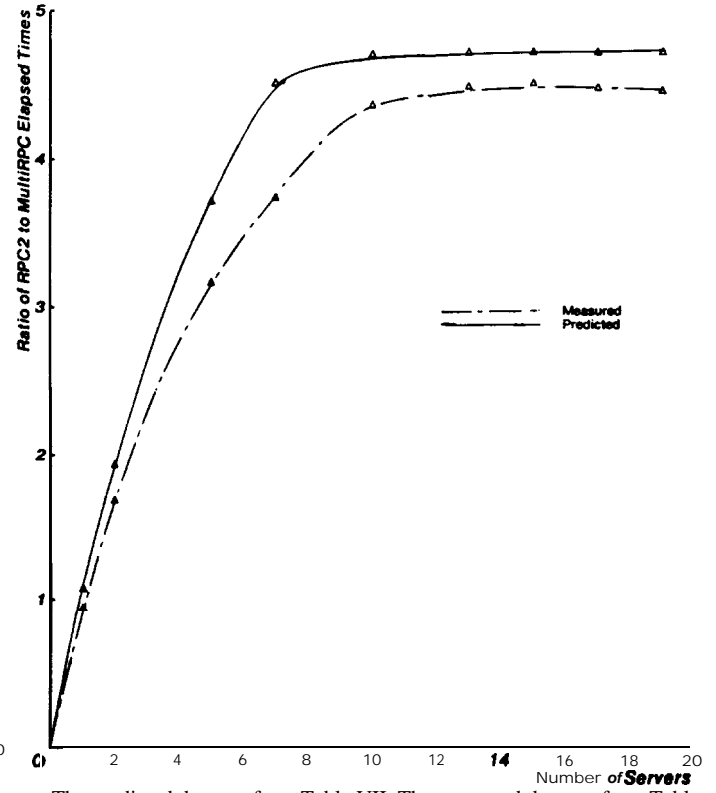
Servers	10 ms Computation			20 ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	37.68 (3.98)	36.80 (6.32)	0.98	44.96 (5.14)	44.64 (5.42)	0.99	91.48 (55.27)	85.20 (51.35)	9.93
2	45.63 (4.14)	72.80 (5.35)	1.53	62.84 (7.71)	91.96 (8.97)	1.46	111.44 (63.78)	158.96 (73.82)	1.43
5	12.92 (4.15)	206.48 (14.25)	2.83	94.12 (7.01)	250.64 (27.30)	2.66	150.92 (46.55)	395.52 (120.55)	2.62
10	111.20 (2.17)	439.65 (22.78)	3.95	123.96 (7.71)	521.96 (80.40)	4.21	227.96 (92.85)	788.36 (115.91)	3.46
15	162.90 (1.52)	626.70 (53.22)	3.85	169.20 (12.97)	906.55 (82.43)	5.36	249.80 (60.33)	1203.56 (237.48)	4.82
19	211.95 (12.15)	843.95 (54.43)	3.98	*	*	*	*	*	*
20	*	*	*	210.05 (13.55)	1182.65 (108.17)	5.63	288.90 (72.33)	1654.80 (232.17)	5.73
25	*	*	*	256.60 (20.49)	1381.30 (88.92)	5.38	304.92 (78.89)	2175.48 (307.59)	7.13
50	768.20 (10.75)	2835.90 (170.28)	3.69	787.05 (14.77)	3166.40 (146.35)	4.02	812.60 (110.29)	4559.30 (282.20)	5.61
75	1358.11 (139.52)	4390.60 (94.70)	3.23	1227.73 (148.51)	5272.80 (462.43)	4.29	1015.63 (38.86)	7543.70 (521.72)	7.43
100	1675.63 (39.54)	607490 (118.74)	3.62	1586.40 (52.27)	7335.90 (419.85)	4.62	1518.90 (212.51)	10724.90 (2397.54)	7.06

All times are in milliseconds. Figures in parentheses are standard deviations. These data were obtained from servers distributed over many network segments. A subset of these data is graphically presented in Fig. 14. Each data point was obtained from 10 trials.

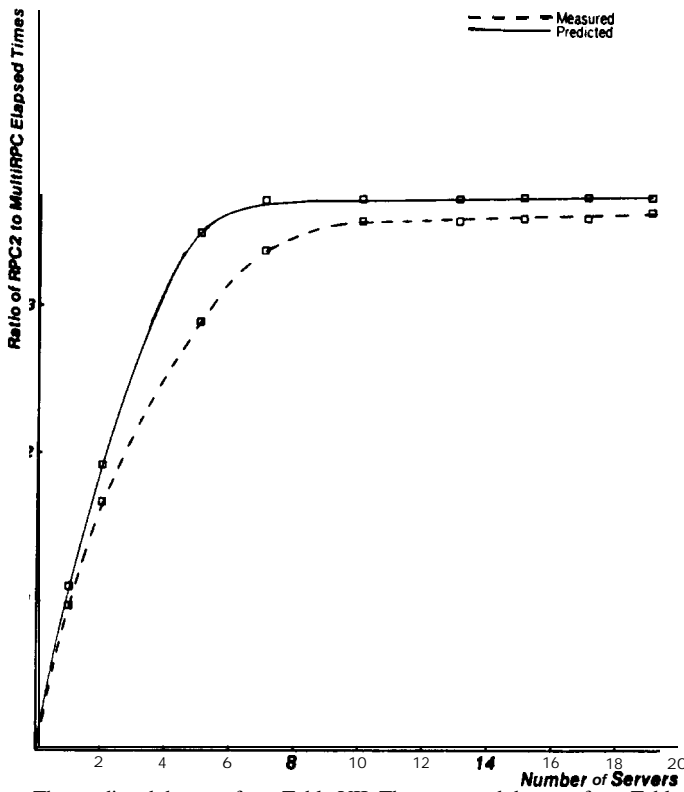
APPENDIX V
GRAPHS



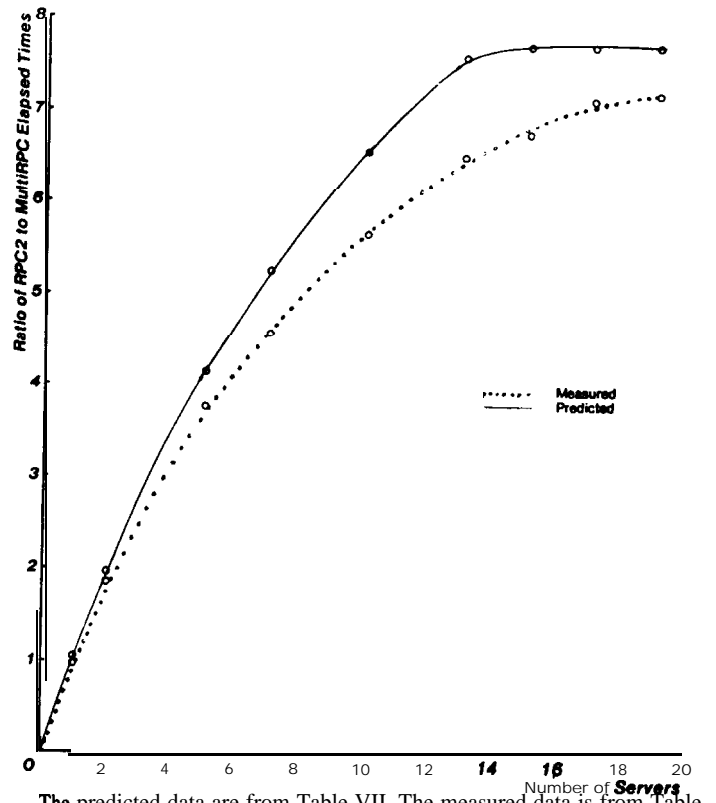
The data in this graph are obtained from Table VII.
Fig. 8. Predicted performance from model (constant computation time).



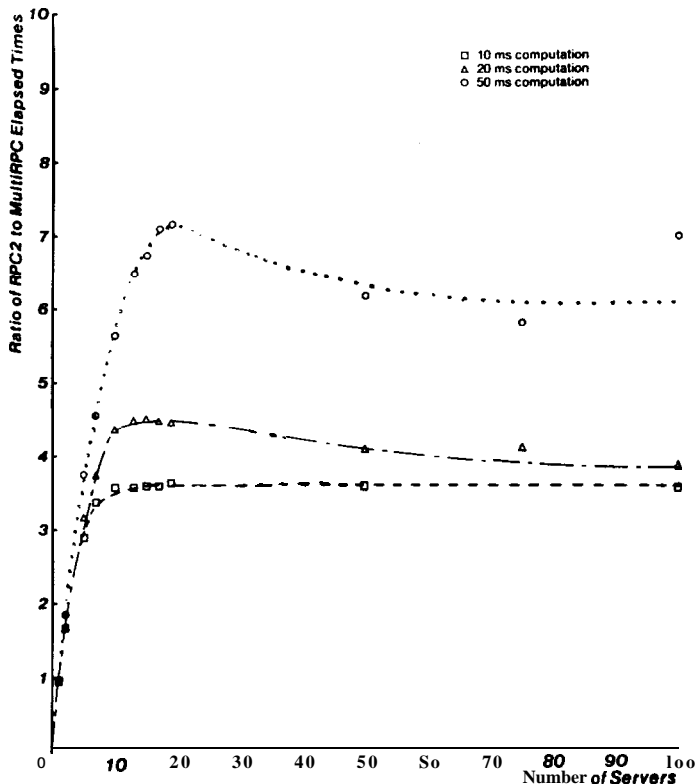
The predicted data are from Table VII. The measured data are from Table VIII.
Fig. 10. Comparison of predicted and measured performance (20 ms computation).



The predicted data are from Table VII. The measured data are from Table VIII.
Fig. 9. Comparison of predicted and measured performance (10 ms computation).

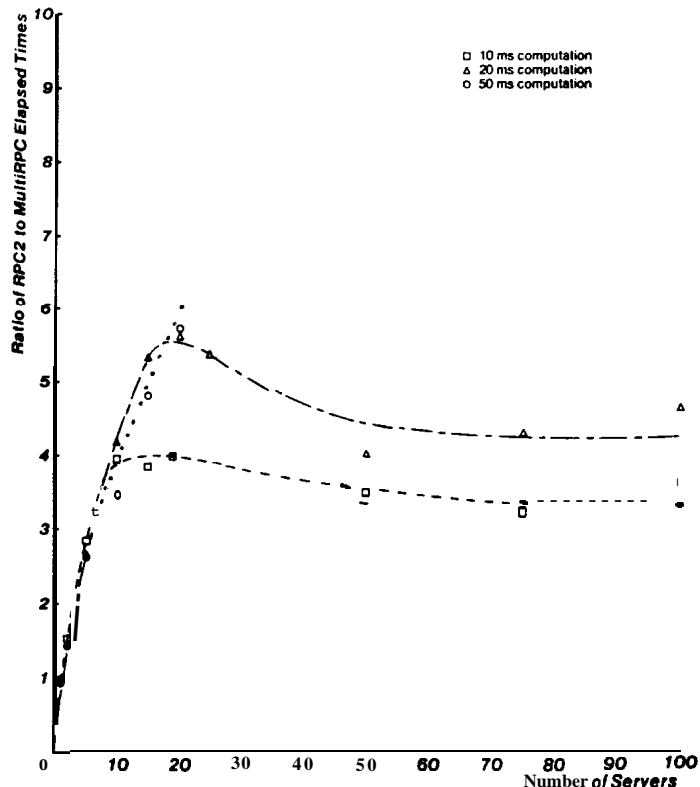


The predicted data are from Table VII. The measured data is from Table VIII.
Fig. 11. Comparison of predicted and measured performance (50 ms computation).



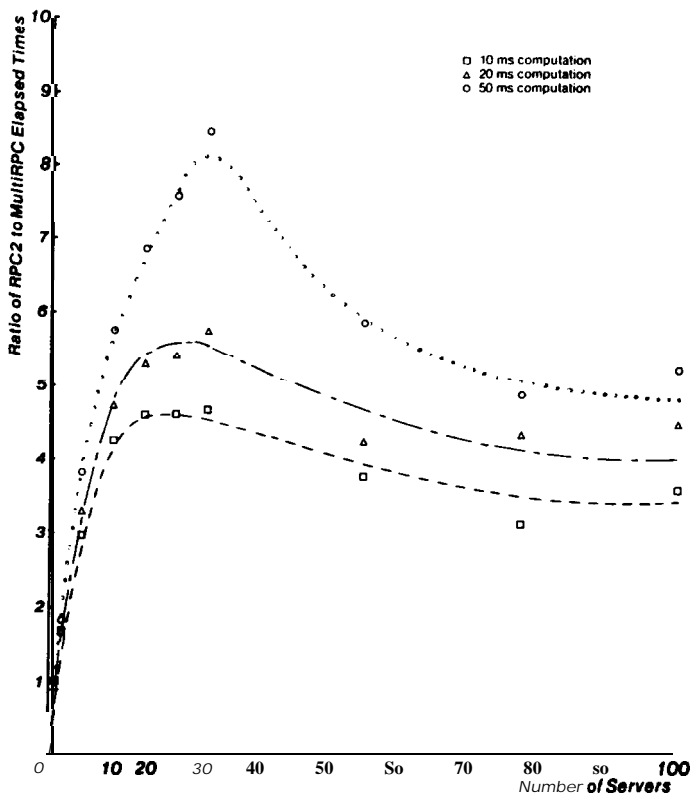
The data in this graph are obtained from Table VIII. Section VI-B and Table VIII describe the conditions under which these data were obtained. The data beyond 50 servers has substantially greater standard deviations than the data for 50 or fewer servers.

Fig. 12. Measured performance for constant server computation time.



The data in this graph are obtained from Table X. Section VI-B and Table X describe the conditions under which these data were obtained. The data beyond 50 servers has substantially greater standard deviations than the data for 50 or fewer servers.

Fig. 14. Measured performance for exponential server computation time.



The data in this graph are obtained from Table IX. Section VI-B and Table IX describe the conditions under which these data were obtained. The data beyond 50 servers has substantially greater standard deviations than the data for 50 or fewer servers.

Fig. 13. Measured performance for normal server computation time.

ACKNOWLEDGMENT

We would like to express our appreciation to J. Kistler, D. Duchamp, B. White, and E. Cooper for their careful reading of this paper and their many useful comments. T. Holodnik of the Data Communications Group at Carnegie Mellon University provided us with Fig. 7. The referees provided us with many good suggestions for improving the paper.

REFERENCES

- [1] A. D. Birrell, A. D. Levin, R. M. Needhan, and M. D. Schroeder, "Grapevine: An exercise in distributed computing," *Commun. ACM.*, vol. 25, no. 4, pp. 260-274, Apr. 1982.
- [2] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39-59, Feb. 1984.
- [3] W. A. Burkhard, B. E. Martin, and J. F. Paris, "The Gemini replicated file system test-bed," in *Proc. Third Int. Conf. Data Eng.*, 1987.
- [4] D. R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Trans. Comput. Syst.*, vol. 2, no. 2, pp. 77-107, May 1985.
- [5] E. C. Cooper, "Replicated distributed programs," in *Proc. Tenth ACM Symp. Oper. Syst. Principles*, Dec. 1985.
- [6] Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC 791: *Internet Program Protocol Specification*, Sept. 1981.
- [7] Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC 768: *User Datagram Protocol Specification*, Sept. 1981.
- [8] Defense Advanced Research Projects Agency Information Processing Techniques Office, RFC 793: *Transmission Control Protocol Specification*, 1981.

- [9] M. Herlihy, "A quorum-consensus replication method for abstract data types," *ACM Trans. Comput. Syst.*, vol. 4, no. 1, pp. 32-53, Feb. 1986.
- [10] M. B. Jones, R. F. Rashid, and M. Thompson, "MatchMaker: An interprocess specification language," in *Proc. ACM Conf.*, Jan. 1985.
- [11] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. E. Weihl, "Communication in the Mercury system," in *Proc. Twenty-First Annu. Hawaii Int. Conf. Syst. Sci.*, Jan. 1988.
- [12] B. Martin, "Parallel remote procedure call language reference and user's guide," *Comput. Syst. Res. Group*, Univ. of California, San Diego, CA 1986.
- [13] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A distributed personal computing environment," *Commun. ACM*, vol. 29, no. 3, pp. 184-201, Mar. 1986.
- [14] J. Rosenberg, L. Raper, D. Nichols, and M. Satyanarayanan, "LWP manual," Inform. Technol. Cent., CMU-ITC-037, 1985.
- [15] J. Sansilo and M. S. Squillante, "An RPC/LWP system for interconnecting heterogeneous systems," in *Proc. USENIX Tech. Conf.*, Dallas, TX, Feb. 1988.
- [16] M. Satyanarayanan, J. H. Howard, D. N. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The ITC distributed file system: Principles and design," in *Proc. Tenth ACM Symp. Oper. Syst. Principles*, Dec. 1985.
- [17] M. Satyanarayanan, "RPC2 user's manual." Inform. Technol. Cent., CMU-ITC038, 1986.
- [18] A. Z. Spector, "Performing remote operations efficiently on a local computer network," *Commun. ACM*, vol. 25, no. 4, pp. 246-260, Apr. 1982.
- [19] Sun Microsystems, Inc., "Networking on the sun workstation," May 15, 1985.



Mahadev Satyanarayanan (S'80-M'83) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1983, after receiving the B.Tech degree in electrical engineering (1975) and M.Tech degree in computer science (1977) from the Indian Institute of Technology, Madras.

He is an Associate Professor of Computer Science at Carnegie Mellon University. He is an experimental computer scientist whose research addresses the problem of data access in large-scale

distributed systems. He is the principal investigator of the Coda project, whose goal is to provide scalable, secure, and highly available file access in a distributed workstation environment. Prior to his work on Coda, he was a principal architect and implementor of the Andrew file system at Carnegie Mellon University. His earlier work included studies of file properties and techniques for modeling storage systems.

Dr. Satyanarayanan is a member of the Association for Computing Machinery, and Sigma Xi. In 1987, he was made a Presidential Young Investigator by the National Science Foundation.



Ellen H. Siegel received the S.B. degrees in chemistry and computer science from the Massachusetts Institute of Technology and the M.S. degree in computer science from Carnegie Mellon University, Pittsburgh, PA.

She is currently working toward the Ph.D. degree in computer science at Carnegie Mellon University, Pittsburgh, PA. Her research interests include distributed systems, communications, and fault tolerant computing.