CMU-ITC-86-047

MultiRPC: A Parallel Remote Call Mechanism

CMU-ITC-047
2 August 1986

(also issued as CMU-CS-86-139)

M. Satyanarayanan
Ellen H. Siegel

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Abstract

RPC2 is a remote procedure call mechanism that has been extensively used in the Andrew system at Carnegie Mellon University. Andrew will eventually encompass many thousands of workstations sharing a single distributed file system. *MultiRPC* is an extension to RPC2 that enables a client to simultaneously perform remote invocations of multiple servers. The concurrency of processing on the servers, and the overlapping of the retransmissions and timeouts with respect to each server result in parallelism that is useful in a number of applications. A key property of MultiRPC is that the exactly-once semantics of RPC2 is retained on each of the parallel calls. In addition, all the basic functionality of RPC2, such as secure, authenticated communication and the use of application-specific side effects, is retained in MultiRPC. The underlying communication medium does not have to support multicast or broadcast transmissions. In this paper we describe MultiRPC and show how we have made it versatile and simple to use without compromising runtime efficiency.

Table of Contents

1. Introduction

RPC2 is a remote procedure call mechanism that has been extensively used in the Andrew distributed computing environment at Carnegie Mellon University [7]. A detailed description of RPC2 may be found elsewhere [11, 10]. *MultiRPC* is an extension to RPC2 that enables a client to perform remote invocations of multiple servers while retaining the exactly-once semantics of remote procedure calls. In this paper we describe MultiRPC and show how we have made it versatile and simple to use without compromising runtime efficiency.

Section 2 presents a simplified overview of RPC2. Given this background, Section 3 explains why we extended RPC2 with a parallel invocation mechanism. The considerations that influenced the design of MultiRPC are then put forth in Section 4. Sections 5 and 6 describe the design and implementation of MultiRPC, while Section 7 discusses some of the subtle consequences of our design decisions. The results of controlled experiments evaluating the performance of MultiRPC are reported in Section 8. Sections 9 relates this work to other efforts aimed at parallelism in network communication. Section 10 concludes the paper with a description of the current status of MultiRPC.

2. An Overview of RPC2

RPC2 consists of two relatively independent components: a Unix-based runtime library written in C, and a stub generator, *RP2Gen*. The runtime 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 procedure calls that make up a remote interface. If one views a remote interface as an abstract data type, the set of operations on that data type constitute a subsystem. RP2Gen takes a description of a subsystem and automatically generates code to marshall and unmarshall parameters in the client and server stubs. It thus performs a function similar to Lupine in the Xerox RPC mechanism [1] and Matchmaker in Accent IPC [6].

The RPC2 runtime system is fully integrated with a Lightweight Process mechanism (*LWP*) [8] 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 LWPs 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 standard Unix. Both RPC2 and the LWP package are entirely outside the Unix kernel and have been ported to multiple machine types.

The low-level packet transport mechanism is a separable component of RPC2. The only primitives required of it are the ability to send and receive datagrams. At present, RPC2 runs on the DARPA IP/UDP protocol [3, 4].

RPC2 is based on logical *Connections*. The rationale for choosing a connection-based rather than connectionless protocol is presented in the design document [11]. For the purposes of this paper the following facts about RPC2 connections are relevant:

1. A connection is created when a client invokes the *RPC2_Bind* primitive and is destroyed by the *RPC2_Unbind* primitive. The cost of an RPC2_Bind is quite low, and is comparable to the cost of a normal RPC.

2. One can view RPC2_Bind as a special RPC that is common to all subsystems. In fact, a server is notified of the creation of a new connection in exactly the same way it is notified of an RPC on an existing connection.

3. Connections use little storage. Typically a connection requires a hundred bytes at each of the client and server ends.

4. Within each Unix process, an RPC2 connection is identified by a unique *Handle*. Handles are never reused during the life of a process.

5. At any given time a Unix process can have at most 65536 active connections. This is about two orders of magnitude larger than the number of connections in use in the most heavily used 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. On a given connection, however, the roles of the peers are fixed.

Besides RPC2_Bind and RPC2_Unbind, the most important runtime primitive on the client side is *RPC2_MakeRPC*. This call sends a contiguous request packet to a server and then awaits a contiguous reply packet. Reliable delivery is guaranteed by a retransmission protocol built on top of the datagram transport mechanism. Calls may take arbitrarily long: keep-alive packets (*BUSY* packets) are sent by the server during a call to inform the client that it is still alive and connected to the network. When the reply is finally received, the server is sent an acknowledgement. As an optimization, the acknowledgement is often piggy-backed on the next call to the server. On the server side, the basic primitives are *RPC2_GetRequest*, which blocks until a request is received and *RPC2_SendResponse*, which sends out a contiguous reply packet and awaits positive acknowledgement of its receipt. Thus RPC2 provides *Exactly-Once* semantics in the absence of site and hard network failures, and *At-Most-Once* semantics otherwise [12].

A unique aspect of RPC2 is its support of arbitrary *Side Effects* on RPC calls. Associated with every RPC connection is a (possibly null) side effect type, specified by the client at the time the connection is made. The side effect mechanism allows application-specific protocols to be integrated with the base RPC2 code. In the Andrew file system, for example, files are transferred via side effects using a highly pipelined protocol. In this application, a special protocol is called for because of the large quantity of data transmitted. The integration of the file transfer protocol into RPC2 does, however, allow small files to be piggybacked on normal RPC packets, thus minimizing the number of packets exchanged in this important special case. Information specific to a side effect is provided by the client and server in *Side Effect Descriptors.* The client provides a side effect descriptor as a parameter to the RPC2_MakeRPC primitive. On the server side, *RPC2_InitSideEffect* initiates a side effect and *RPC2_CheckSideEffect* awaits its completion. Each of these primitives takes a side effect descriptor as a parameter.

The design documents [11, 10] describe a number of other RPC2 capabilities, such as the use of *Filters* to allow flexibility in structuring a server, and the incorporation of a hierarchy of *Security Levels* based on authentication and encryption. We omit a discussion of those topics here since they are not essential to an understanding of MultiRPC. Tables 2-1, 2-2, and 2-3 summarize the RPC2 primitives relevant to this paper.

Primitive	**Description**
RPC2_Bind ( )	Create a new connection
RPC2_MakeRPC ( )	Make a remote procedure call (with possible side effect)
RPC2_MultiRPC ( )	Make a collection of remote procedure calls

Table 2-1: Client Related Primitives

Primitive	**Description**
RPC2_Export ( )	Indicate willingness to accept calls for a subsystem
RPC2_DeExport ( )	Stop accepting new connections for one or all subsystems
RPC2_GetRequest ( )	Wait for an RPC request or a new connection
RPC2_Enable ( )	Allow servicing of requests on a new connection
RPC2_SendResponse ( )	Respond to a request from my client
RPC2_InitSideEffect ( )	Initiate side effect
RPC2_CheckSideEffect ( )	Check progress of side effect

Table 2-2: Server Related Primitives

3. Motivation for MultiRPC

The principles underlying MultiRPC arose as a solution to a specific problem in Andrew. In the Andrew file system [9], workstations cache files on their local disks from servers. 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

Primitive	Description
RPC2_Init ( )	Perform runtime system initialization
RPC2_Unbind ( )	Terminate a connection by client or server
RPC2_AllocBuffer ( )	Allocate a packet buffer
RPC2_FreeBuffer ( )	Free a packet buffer

Table 2-3: Miscellaneous Primitives

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 utimately 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 has to 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 feasible. 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 packet were relevant to that workstation. Using multicast to narrow the set of workstations contacted is also impractical, because each file would then potentially 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.

Besides these considerations, the use of broadcast or multicast does not provide servers with confirmation that individual workstations have indeed received the callback information. Such confirmation is implicit in the reliable delivery semantics of RPC2. It became clear to us that we needed a mechanism that retained the strict RPC semantics of RPC2 while overlapping the computation and communication overheads at each of the destinations. This is the essence of MultiRPC.

MultiRPC has applications in other contexts too. Replication algorithms such as quorum consensus [5] 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.

The performance of some relatively simple but frequent operations in large distributed systems may also be improved by MultiRPC. Consider, for example, the contacting of a name or time server. If more than one such server is available, it may be reasonable to use MultiRPC to contact many of them, wait for the earliest reply and abandon all further replies. Another example, relevant to Andrew, is the use of MultiRPC in monitoring the status of multiple file servers. The monitoring program periodically contacts all the servers and requests status information from each of them.

4. Design Considerations

The primary consideration in the design of MultiRPC was that it be inexpensive. Normal RPC calls should not be slowed down because of MultiRPC. Although one-to-many RPC calls constitute a very important special case, we still expect simple, one-to-one RPC calls to be preponderant. A related, but distinct, concern is the increase in program size resulting from MultiRPC. Virtual memory usage in our workstations is considerably higher than we originally anticipated. We wished to minimize the impact of MultiRPC on this problem.

Another influence on our design was the desire to decouple the design of subsystems from considerations relating to MultiRPC. No special coding should be necessary in order to make a subsystem available via MultiRPC. Similarly, clients who access that service via simple RPC calls should be unaffected. Only those clients who wish to access that subsystem in parallel at multiple sites should have to know about MultiRPC. In addition we wished to have the flexibility of intermixing simple RPC and MultiRPC calls in any order on any combination of connections.

Since we did not wish to distinguish between connections usable in MultiRPC calls and those usable only in simple RPC calls, MultiRPC has to be completely orthogonal to normal RPC2 features. The exactly-once delivery semantics, failure detection, support for multiple security levels and the ability to use side effects all have to be retained when making a MultiRPC call.

A number of the scenarios in which we envisage MultiRPC being used require replies to be processed by the client as they arrive, rather than being batched for final processing. The exact nature of such processing is application-dependent and therefore has to be performed by a client-specified procedure. In addition, we felt it was important that a client be able to abort the MultiRPC call after processing any of the replies or after a predetermined amount of time has elapsed since the start of the MultiRPC call.

Finally, we wanted MultiRPC 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, the latter being similar to a local procedure call. We have had to violate this syntax for two reasons: to allow clients to specify an arbitrary reply-handling procedure in a MultiRPC call, and to avoid expanding code size by generating a MultiRPC stub for every call in a subsystem.

5. Using MultiRPC

We begin our description of MultiRPC with an example. 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 designs 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.

Figure 5-1 presents the specification of the subsystem *example* in the file *example.rpc2*. RP2Gen interprets this 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 Figure 5-2). This subsytem is composed of two operations, *double* and *triple*. 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.

```
RP2Gen specification file for simple subsystem

Server Prefix "serv";                          tag server operations to avoid ambiguity

Subsystem "Example";

double(IN OUT RPC2_Integer testval);
triple(IN OUT RPC2_Integer testval);
```

Figure 5-1: *example.rpc2* specification file

Once the interface has been specified, the subsystem implementor 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. Figures 5-3 and 5-4 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*, used by MultiRPC to marshall and unmarshall 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 Figures 5-5 and 5-6.

```
/*
    (c) Copyright IBM corporation, 1985

    .h file produced by RP2GEN

    Input file: ex.rpc2
*/
#include "rpc2.h"
#include "se.h"

/* Op codes and definitions */

#define double-OP    1
extern ARG double-ARGS[];
#define double-PTR     double-ARGS

#define triple-OP     2
extern ARG triple-ARGS[];
#define triple-PTR     triple-ARGS
```

Figure 5-2: The RP2Gen-generated header file *example.h*

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 *MRPC_MakeMulti* is used to access the runtime system routine *RPC2_MultiRPC*. The use of a common library routine rather than individual stubs cuts down on code size but does require some additionalinformation from the client making the call.

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

6. Design and Implementation

Since MultiRPC was intended to fit into the existing RPC2 package, its design mirrors that of RPC2. Support for MultiRPC exists both at the runtime level and at the language level. MultiRPC is supported by one RPC2 runtime primitive, RPC2_MultiRPC, and the RPC2 library routines MRPC_MakeMulti and *MRPC_UnpackMulti* which perform the parameter marshalling and unmarshalling. Language support for MultiRPC has been achieved by extending RP2Gen to generate argument descriptor structures for each server operation.

Although MultiRPC retains the semantics of an RPC2 call, its syntax diverges slightly from a strict procedural interface. The call no longer resembles direct invocation of the server operation, but is made via

```
Trivial client illustrating RPC2 call format

main()
{
 int testval;

Perform LWP and RPC2 Initialization

Establish connections to server using RPC2_Bind:

 RPC2 - Bind(/* Bind arguments */);

while (TRUE)
    {
    printf("\nDouble [ = 1] or Triple [ = 2] (type 0 to quit)? ");
    scanf("%d", op);
    if (op = = 0)
         break;
    printf(Number? ");
    scanf("%d", testval);
    if (op = = 1) {
     double(cid, testval);              Perform RPC2 call on connection cid
     printf("doubled value = %d\n", testval);
     }
    else {
     triple(cid, testval);
     printf("tripled value = %d\n", testval);
     }
    }

 printf("Bye...\n");
 RPC2 - Unbind(cid);
}
```

Figure 5-3: A Simple RPC2 Client

MRPC_MakeMulti. A client can perform a MultiRPC call on any subsystem with an RP2Gen generated interface, but the MultiRPC packing and unpacking facilities exist in a pair of library routines rather than in a stub package as in RPC2. The slight additional processing cost of parameter interpretation is outweighed by the savings in the code size. The packet which is finally transmitted to the server is identical to a packet generated by an RPC2 call, and the MultiRPC protocol requires only a normal response from each server. A server is not even aware that it is participating in a MultiRPC call.

The following routines form the backbone of MultiRPC.

RPC2_MultiRPC is the parallel analog of the RPC2_MakeRPC call. It initializes the RPC2 runtime state in preparation for packet transmission and then calls *mrpc_SendPacketsReliably*, an internal routine which performs the request transmissions and processes the server responses.

Server loop for example RPC2 subsystem

```
main()
  {
  RPC2_RequestFilter  reqfilter;
  RPC2_PacketBuffer *reqbuffer;
  RPC2_Handle cid;

  InitRPC();                          Perform LWP and RPC initialization
                                      Set filter to accept auth requests on new or existing connections

  while(TRUE)
    {
                                      Await a client request
    RPC2_GetRequest(&reqfilter, &cid, &reqbuffer, NULL, NULL, NULL, NULL);
    serv-ExecuteRequest(cid, reqbuffer)         This routine is generated by RP2Gen
    }
  }
```

Trivial server procedure to demonstrate RPC2 functionality

```
long serv-double(cid, testval)
     RPC2-Handle cid;
     RPC2-Integer *testval;
     {
     *testval = (*testval) * 2;
     }


long serv-triple(cid, testval)
     RPC2-Handle cid;
     RPC2-Integer *testval;
     {
     *testval = (*testval) * 3;
     }
```

Figure 5-4: A Simple RPC2 Server

MRPC_MakeMulti

is the library routine which provides the parameter packing interface to RPC2_MultiRPC. It takes as arguments RP2Gen generated argument descriptor structures, the appropriate opcode (from the *<subsys>.h) file*) the number of servers to be called, and a pointer to a client supplied handler routine as well as an array of connection ids and the appropriate server arguments. Using the argument descriptors, MRPC_MakeMulti packs the supplied server arguments into an RPC2 request buffer; it then creates a data structure containing argument descriptor structures and a pointer to the client handler routine. Finally, it makes the RPC2_MultiRPC call and passes the final return code back to the client when the call terminates.

MRPC_UnpackMulti

is a RPC2 library routine which functions as the complement of MRPC_MakeMulti. It

```
The following demonstrates a simple MultiRPC call

Include RPC2 and any other include files

#define HOWMANY 3
long HandleResult();

main()
{
 int testval[MAXSERVERS];
 RPC2_Handle cid[MAXSERVERS];

Perform LWP and RPC2 Initialization
Establish connections to all desired servers using RPC2_Bind

 for(count = 0; count < HOWMANY; count ++) {
    ret = RPC2-Bind(/* Bind arguments */);
 }

 while (TRUE)
    {
    printf("\nDouble [ = 1] or Triple [ = 2] (type 0 to quit)? ");
    scanf("%d", op);
    if (op == 0) break;
    printf("\nNumber? ");
    scanf("%d", testval);
Make the MultiRPC call
    if (op == 1)
     MRPC_MakeMulti(double-OP, double-PTR, HOWMANY, cid, HandleDouble, NULL, testval);
    else MRPC_MakeMulti(triple-OP, triple-PTR, HOWMANY, cid, HandleTriple, NULL, testval);
    printf("results = %d, %d, %d\n", testval[0], testval[1], testval[2]);
    }

 printf("Bye...\n");
 for (count = 0; count < HOWMANY; count++)
   RPC2-Unbind(cid);
}
```

Figure 5-5: Client using MultiRPC

unpacks the contents of the response buffer into the appropriate positions in the client's argument list and calls the client handler routine. It returns with the return code supplied by the client handler routine.

mrpc_SendPacketsReliably

is an internal RPC2 runtime routine which is the heart of MultiRPC retransmission and result gathering. It is called by RPC2_MultiRPC with the prepared request buffers, performs an initial transmission of all the request packets, and then suspends itself to wait for the server responses. As each response arrives, RPC2 error codes are checked and any appropriate side effect processing is performed. mrpc_SendPacketsReliably then calls MRPC_UnpackMulti with the server reply buffer and the relevant return code. Any client-specified timeout will expire in this routine.

Client Supplied Handler Routines

```
long HandleDouble(HowMany, ConnArray, offset, rpcval, testval)
RPC2 - Integer HowMany, offset, rpcval, testval[];
RPC2 - Handle ConnArray[];
{
 if (rpcval != RPC2 - SUCCESS)
     printf("HandleResult: rpcval = %d\n", rpcval);
 if(+ +returns = = HOWMANY) return -1;       Terminate the MultiRPC call
 return(0);                                  Continue accepting server responses
}

long HandleTriple(HowMany, ConnArray, offset, rpcval, testval)
RPC2 - Integer HowMany, offset, rpcval, testval[];
RPC2 - Handle ConnArray[];
{
 if (rpcval != RPC2 - SUCCESS)
     printf("HandleResult: rpcval = %d\n", rpcval);
 if(+ +returns = = HOWMANY) return -1;       Terminate the MultiRPC call
 return(0);                                  Continue accepting server responses
}
```

Figure 5-6: MultiRPC Client Handler Routines

The client supplied handler routine is called once for each connection by MRPC_UnpackMulti either with the newly arrived server reply or with connection failure information. The handler routine allows the client to process each server response as it arrives rather than waiting for the entire MultiRPC call to complete. After processing each response, the client can decide whether to continue accepting server responses or whether to abort the remainder of the call. It can perform as much or as little processing as the client deems necessary.

The library packing routines receive their call-specific information from argument descriptor structures created by RP2Gen and passed to MultiRPC by the client. RP2Gen generates an array of these descriptors for each server operation, with one structure being generated for each parameter. The structure fields contain information about the parameter's type, usage (e.g. IN, OUT, IN - OUT) and size; for structure arguments, the descriptor also points to an array of descriptors describing the structure fields. The argument packing and unpacking is performed at runtime by iterating in parallel through the descriptors and the argument list. Since structures can be nested arbitrarily deeply, structure packing is done recursively using nested descriptors.

MultiRPC provides the same correctness guarantees as the standard RPC2 package except in the case where the client exercises his right to terminate the call. If the call completes normally, a return code of RPC2_SUCCESS guarantees that all messages have been received by the appropriate servers.

7. Failures and Error Conditions

The semantics for errors in the MultiRPC case are somewhat different from those in RPC2. Since several messages are being transmitted in the same call, an error on one connection should not be sufficient to terminate the call. The client does, however, need to be informed of error states on any of his connections. The handler routine will be called at most once for each connection submitted to the MultiRPC call, either with an error condition or with the server response. No packet will be sent on any connection for which an error was detected in the course of initial processing.

The additional flexibility provided by the client handler routine incurs some risks. RPC2 makes no guarantees about the state of the connections which have not been examined before a client-initiated termination of MultiRPC. When the client returns an abort code from his handler, there may still be replies outstanding on some connections. On each such connection RPC2_MultiRPC increments the connection sequence number and resets the connection state, thus pretending that the response in question was actually received. This allows the system to continue with normal operation.

If the response in question does arrive before the next call is made it will be thrown away as an old response and an acknowledgement will be sent to the server. If, however, it arrives after a second call has been aborted by the client it will be treated as a bad packet and thrown away without an acknowledgement. Further requests will now be thrown away because the server will be repeatedly retrying the original response and waiting for an acknowledgement. The server will eventually decide that the client is dead, but will waste retrying before it gives up. The client can continue indefinitely without realizing that the server is functionally dead. The client will not appreciate his predicament unless he waits for responses on all connections in the MultiRPC call or sends a simple RPC2 call on that specific connection.

This pathological situation was observed during initial testing of MultiRPC, and led to a change in the implementation of RPC2. If a packet whose sequence number is greater than that expected by its connection state arrives, an explicit negative acknowledgement is now sent. The negative acknowledgement will kill the connection. This allows the client to learn explicitly that the connection is useless and give him an opportunity to rebind to the relevant server. It also lets the server recognize the failure promptly.

These problems are a result of the client's ability to ignore the responses on some connections in a MultiRPC call, and will generally only manifest themselves in a case where a server is forced to queue a request because it is busy processing an earlier request. This means that the MultiRPC call should be used with caution in cases where simultaneous binding to a single site might result. The negative acknowledgement by the server is intended to give the client the opportunity to learn that something has gone wrong with the connection and act accordingly.

8. Performance

In this section we assess MultiRPC performance by focussing on two specific questions. First, how is the RPC roundtrip time to a single site affected by using MultiRPC rather than RPC2? Second, as a function of the number of sites contacted, how much reduction in elapsed time does one obtain by using MultiRPC?

Our analysis is based on actual controlled experimentation. The experiments were conducted on unloaded IBM RT-PC workstations running the Unix 4.2BSD operating system. Some of these workstations are on an Ethernet while others are attached to an IBM token ring network. Routers provide the necessary connectivity between all these workstations. The experiments were run at periods of low network utilization.

Table I-1 shows the effect of using MultiRPC to contact a single site. Ten trials, each consisting of 1000 null RPC2 calls and 1000 null MultiRPC calls, were run. As the table shows, the difference in elapsed times for these two cases is negligible. Further, the null RPC2 times are within 3 percent of the times we observed on an earlier version of RPC2 that lacked MultiRPC support. Our original design criterion of not slowing down simple RPCs has thus been met.

The server computation time per RPC request is a key parameter in determining the improvement due to MultiRPC. If this time is very short, the transmission, marshalling and unmarshalling overheads dominate. If the time is very long, these overheads are negligible and the overlapping of server computation improves performance considerably.

We measured the effect of varying the number of servers and the length of the server computation time on the performance of MultiRPC. These tests were for 1 to 50 servers with computation times in the range 10 milliseconds to 1 second. The data from these experiments is presented in tabular form in Tables I-2 through I-5 and graphically in Figure II-1. In the tables and graphs the roundtrip elapsed time for a MultiRPC call is compared with the elapsed time for the equivalent number of iterative RPC2 calls.

As expected, the performance improvements are larger for longer computation times. For RPC2, these computation times are strictly additive; for MultiRPC, the server computation times are overlapped significantly. One would typically expect the performance to improve monotonically for increasing numbers of servers. Figure II-1 shows, however, that beyond about 20 servers the performance saturates and in fact declines slightly. Detailed examination of the experimental data revealed an increase in the number of retried packets beyond about 20 servers. This coincidence led us to the hypothesis that the constant computation time at the servers is the origin of the problem. All the servers send their responses back at the same time, saturating the client with packets and causing it to drop many of them.

To test our hypothesis we repeated the experiments with the computation times exponentially distributed around a specified mean. The data for these tests is presented in Tables I-6 through I-9 and in Figure II-2. For small computation times these measurements still show that saturation occurs. However, for a 1 second computation time performance monotonically improves up to 50 servers.

To obtain more conclusive evidence and to further investigate the performance characteristics of MultiRPC, we are planning experiments with a larger number of servers and with different server computation time distributions. Measurements from actual applications will provide us with further insights into MultiRPC performance.

9. Related Work

Work in the area of parallel network access has typically addressed broadcast and multicast protocols. The Sun Microsystems Broadcast RPC [13] has a design similar to that of MultiRPC, but chooses a low-level broadcast mechanism. The Sun Broadcast RPC resolves addresses by going through a central port, so servers must register with the central port in order to be accessible via the broadcast facility. This means that the server must allow in advance for the possibility of broadcast communication, rather than leaving the decision completely in the hands of the client. MultiRPC's connection-based communication allows parallel calls to be made to any existing servers.

Sun Broadcast RPC provides the client with the flexibility of a timeout and the equivalent of MultiRPC's client handler routine, but it does not provide the same correctness guarantees and error information as MultiRPC. Any unsuccessful response is filtered out and thrown away; no distinction is made between real garbage and response errors such as mismatched version numbers. Since MultiRPC is connection based and orthogonal to the server implementation, all server responses are processed exactly as in the RPC2 case: all relevant error information is recognized and passed on to the client.

The Group Interprocess Communication (IPC) in the V Kernel [2] uses the Ethernet multicast protocol as its basis. This implementation uses the concept of *host groups* as message addresses. A request is multicast throughout the network, and it is up to each host to recognize any group address for which it has local members.

Unlike MultiRPC, reliability is not an important goal of the V group IPC. Multiple retransmissions are suggested to increase reliability, and lost responses are common (due largely to simultaneous arrival of packets at the client). If confirmation is required, it is the client's responsibility to query the appropriate group until he has received adequate confirmation. The client again has control over termination, although the mechanism here is different. The client blocks only until the first response is received, so it can process responses as they arrive. The multicast call is terminated when the client makes another call, causing any further responses to the first call to be thrown away.

Since one of the primary design goals for MultiRPC was retaining the RPC2 correctness guarantees, broadcast and multicast protocols are less attractive despite their transmission efficiency. MultiRPC is heavily dependent on its connection sequence numbers to guarantee its exactly-once semantics, and a multicast protocol would greatly complicate careful sequence numbering. Another issue is security: MultiRPC, like RPC2, allows a per-connection encryption key. This works for MultiRPC because it is a connection based protocol, but site-specific encryption would be impossible for true multicast. In addition, to maintain reliability, acknowledgements would have to be provided. Without connections, this could potentially introduce the lost packet problem of the V group IPC; it would certainly cut down on the performance gains of the multicast protocol itself.

Neither SUN RPC or V Group IPC is able to deal effectively with long computations while providing timely notification of site or network failures. A sufficiently long computation would simply cause a timeout on the connection, indicating server death. MultiRPC, on the other hand, employs a BUSY protocol that alleviates this problem. If a server is busy with a long computation and another request arrives, the low-level RPC2 will queue the request and respond with a BUSY packet so that the client knows that the server is alive. The client then knows to sleep, waking periodically to retry the request. This protocol in conjunction with the LWP package allows useful processing to be performed even when the server is busy with a long computation.

Some of the difficulties of integrating broadcast or multicast with correctness and reliablility could be overcome by application specific protocols with greater domain knowledge. Depending on the application, acknowledgements could possibly be batched at specified intervals; lost packets could be dealt with by high level code. This approach might be useful for specific applications, but it no longer provides the semantics of a general purpose RPC.

10. Status

MultiRPC is fully implemented and operational at the present time. An actual application that now uses MultiRPC is the *Filestats* program. Filestats monitors the status of Andrew file servers by periodically polling them and graphically displaying the results on an operator's console. We will shortly convert the callback mechanism in the Andrew file system to use MultiRPC. At the present time the servers still iterate over RPC2 connections. We also intend to use the MultiRPC facility in implementing a fully replicated version of the Andrew authentication server. In this application we expect MultiRPC to significantly improve the performance of the replication algorithms.

We are aware of at least one potential improvement to MultiRPC. Currently, RP2Gen does not take advantage of repeated argument types when it generates argument descriptor structures, but creates a new descriptor for each parameter of each operation. For recursive structure arguments, these descriptors can become quite large. Since structure arguments must be defined in the RP2Gen specification file, it should be possible to share these descriptors when structure arguments appear more than once.

Using software in actual applications often results in unexpected lessons and refinements to the original design. It is premature at this point in our work to predict what such changes might be. Our immediate future plans involve more detailed and comprehensive testing of MultiRPC. Controlled experimentation and incorporation in application programs will both play a part in making MultiRPC mature, robust and efficient. Based on our experience so far, we are confident that MultiRPC will prove to be an important building block in Andrew and other large distributed systems.

I. Performance Figures for RPC2 versus MultiRPC

The following tables contain roundtrip call times in milliseconds of iterative RPC2 and MultiRPC calls. The figures in parentheses are 90% confidence intervals.

RPC2 (ms)	MultiRPC (ms)	RPC2 / MultiRPC
32.02 (1.93)	32.48 (1.29)	0.986

Table I-1: Times for null RPC2 and MultiRPC calls

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	148.68 (128.96)	152.01 (121.77)	0.978
2	226.49 (235.61)	121.15 (141.44)	1.870
5	192.46 (14.83)	78.81 (4.32)	2.442
10	466.58 (112.76)	169.55 (80.06)	2.752
20	1141.87 (642.42)	385.76 (158.16)	2.960
50	3292.56 (851.39)	1968.84 (945.57)	1.672

Table I-2: 10 ms Computation (Constant)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	79.42 (5.59)	80.66 (12.83)	0.985
2	268.34 (189.24)	155.49 (132.00)	1.726
5	433.37 (95.33)	170.41 (277.90)	2.543
10	943.49 (222.84)	212.97 (278.18)	4.430
20	2237.81 (539.04)	434.91 (338.52)	5.145
50	5534.18 (1155.74)	1830.80 (885.08)	3.023

Table I-3: 50 ms Computation (Constant)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	196.95 (118.57)	194.80 (94.31)	1.011
2	337.27 (128.29)	188.73 (73.96)	1.787
5	667.28 (17.50)	171.68 (3.63)	3.887
10	1418.41 (202.63)	256.98 (123.31)	5.520
20	2933.38 (173.00)	408.12 (338.53)	7.188
50	7770.98 (717.66)	1794.18 (875.09)	4.331

Table I-4: 100 ms Computation (Constant)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	1069.15 (35.55)	1073.16 (97.53)	0.996
2	2116.17 (83.55)	1092.87 (273.58)	1.936
5	5285.91 (121.98)	1103.84 (79.77)	4.789
10	10591.13 (318.89)	1178.58 (116.28)	8.986
20	21248.18 (260.72)	1334.42 (269.60)	15.923
50	53689.08 (NAN ())	3138.50 (802.70)	17.107

Table I-5: 1 second Computation (Constant)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	154.78 (120.58)	138.20 (125.39)	1.120
2	233.13 (164.37)	151.54 (115.45)	1.538
5	450:28 (379.57)	148.46 (146.86)	3.033
10	763.86 (572.55)	209.82 (184.38)	3.641
20	1620.13 (706.83)	439.44 (369.08)	3.687
50	3801.58 (889.03)	1318.58 (757.91)	2.883

Table I-6: 10 ms Computation (Exponential Distribution)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	169.85 (120.41)	153.41 (109.15)	1.107
2	299.60 (170.08)	182.08 (114.42)	1.645
5	629.57 (322.07)	207.38 (116.02)	3.036
10	1182.61 (548.92)	289.44 (193.23)	4.086
20	2212.71 (686.29)	456.04 (327.76)	4.852
50	5843.48 (957.77)	1305.08 (799.55)	4.478

Table I-7: 50 ms Computation(Exponential Distribution)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	182.81 (130.11)	185.31 (132.60)	0.987
2	370.90 (262.26)	244.72 (168.02)	1.516
5	777.25 (356.11)	309.74 (200.25)	2.510
10	1621.05 (620.10)	400.40 (193.64)	4.049
20	3210.61 (752.76)	574.13 (411.51)	5.592
50	7803.74 (1170.27)	1424.02 (775.67)	5.480

Table I-8: 100 ms Computation (Exponential Distribution)

Servers	Iterative RPC2 (ms)	MultiRPC (ms)	Ratio of RPC2/MultiRPC
1	1038.58 (1245.74)	1027.84 (1213.16)	1.010
2	2188.32 (2147.14)	1552.77 (1190.86)	1.409
5	5257.67 (2973.58)	2695.02 (1736.84)	1.951
10	10509.11 (3823.18)	3263.36 (1799.50)	3.220
20	21831.53 (6037.77)	4051.06 (1734.42)	5.389
50	53456.24 (NAN())	4737.28 (1390.78)	11.284

Table I-9: 1 second Computation (Exponential Distribution)

II. Graphs of RPC2 vs MultiRPC Performance

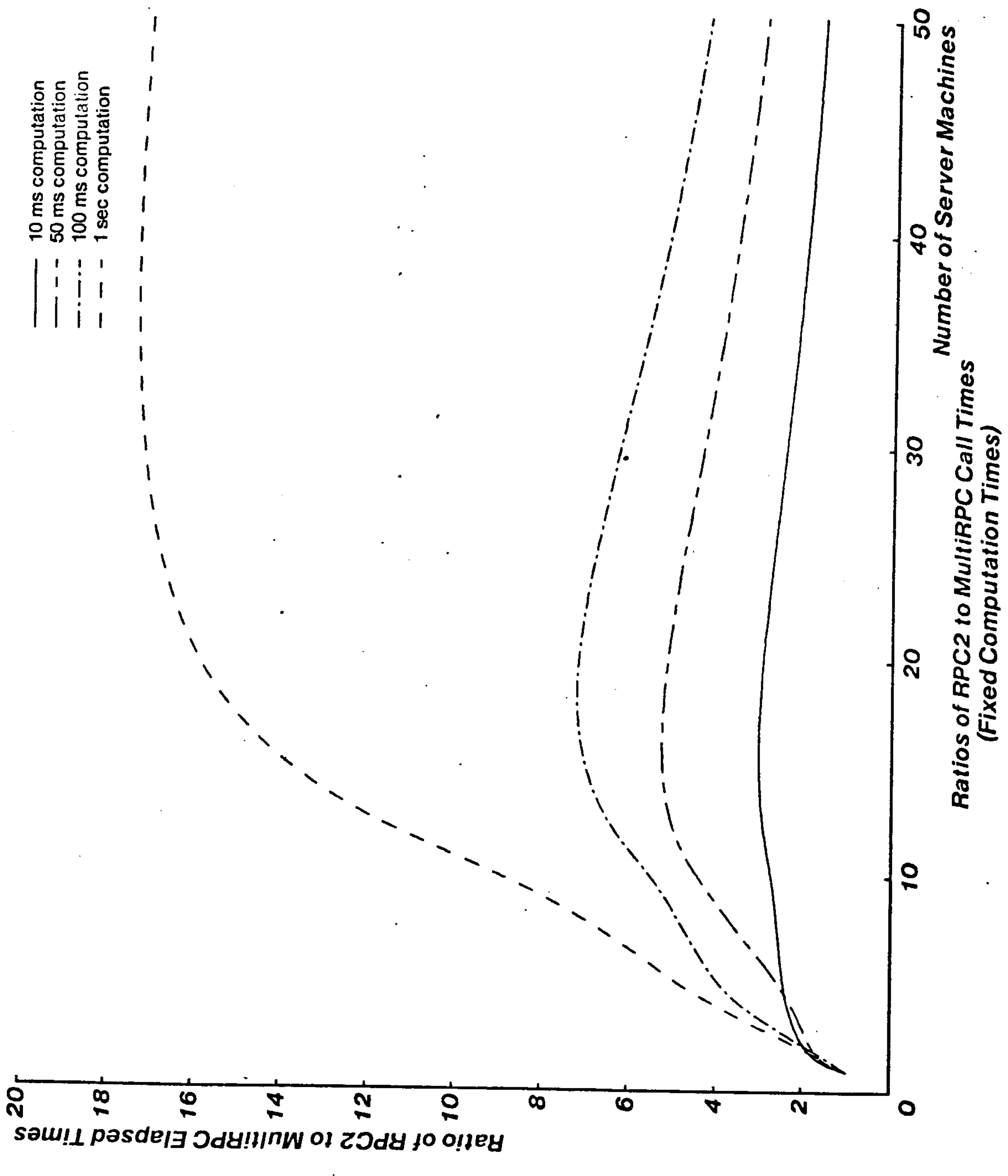

Figure II-1: Constant Computation Time

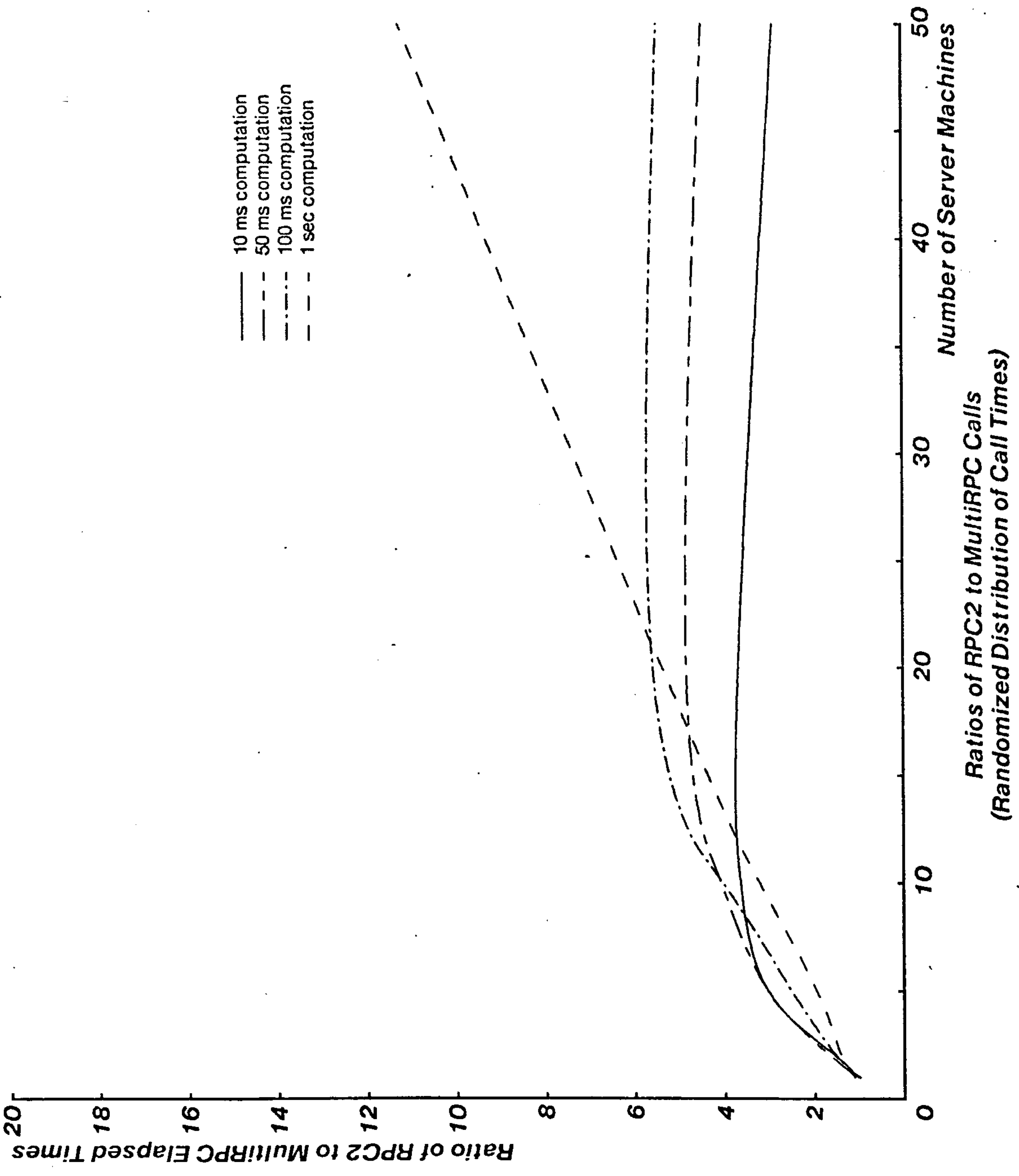

Figure II-2: Exponential Computation Time

References

[1] Birrell, A.D. and Nelson, B.J.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems (1):39-59, February, 1984.

[2] Cheriton, David R., and Zwaenepoel, Willy.
Distributed Process Groups in the V Kernel.
ACM Transactions on Computer Systems 2(2):77-107, May, 1985.

[3] Defense Advanced Research Projects Agency, Information Processing Techniques Office.
RFC 791: Internet Program Protocol Specification
September 1981.

[4] Defense Advanced Research Projects Agency, Information Processing Techniques Office.
RFC 768: User Datagram Protocol Specification
September 1981.

[5] Herlihy, M.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1):32-53, February, 1986.

[6] Jones, M.B., Rashid, R.F., and Thompson, M.
MatchMaker: An interprocess specification language.
In *Proceedings of the ACM Conference on Principles of Programming Languages.* January, 1985.

[7] Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H., and Smith, F.D.
Andrew: A Distributed Personal Computing Environment.
Communications of the ACM 29(3):184-201, March, 1986.

[8] Jonathan Rosenberg, Larry Raper, David Nichols, M. Satyanarayanan.
LWP Manual
Information Technology Center, CMU-ITC-037, 1985.

[9] Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z. and West, M.J.
The ITC Distributed File System: Principles and Design.
In *Proceedings of the 10th ACM Symposium on Operating System Principles.* December, 1985.

[10] M.Satyanarayanan.
RPC2 User Manual
Information Technology Center, CMU-ITC-038, 1986.

[11] M. Satyanarayanan.
RPC2: A Communication Mechanism for Large-Scale Distributed Systems.
Manuscript in preparation, 1986.

[12] Spector, A.Z.
Performing remote operations efficiently on a local computer network.
Communications of the ACM 25(4):246-260, April, 1982.

[13] Sun Microsystems Inc.
Networking on the Sun Workstation
15 May 1985.