REMARKS ON THE COST OF USING A REMOTE PROCEDURE CALL FACILITY

R. Stephen Elliott
Gary J. Nutt

CU-CS-426-89 February, 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

(303) 492-7581
elliott@boulder.colorado.edu
nutt@boulder.colorado.edu

This research has been supported by NSF cooperative agreement DCR-8420944, NSF Grant No. CCR-8802283, and U S West Advanced Technologies.

ABSTRACT

Remarks on the Cost of Using A Remote Procedure Call Facility

Programming a network of computers is a complex exercise. While one aspect of distributed programming is to achieve speed ups as a function of the number of computers used to solve a problem, it is also useful to consider distributed programs in which the distribution of execution is intended to accomplish information sharing across differing domains.

Remote procedure call techniques can be used to accomplish the latter goal. They allow a programmer to access information in remote computers without addressing details of synchronization and interprocess communication. The interface to the remotely executed procedure appears to the programmer just as if it were locally executed.

This paper looks at the performance of a particular implementation of the remote procedure call idea -- the Sun Microsystems implementation. This implementation is a commercial software package intended to be used by application programmers, without concern for the implementation. While the interface to the facilities is well-understood, the potential performance degradation due to remote execution was of concern to us in our utilization of the facility. We report our experimental results with measurements on the Sun Remote Procedure Call facility.

While we found that serious inefficiencies could result in cases where the facility was "automatically" accomplishing synchronization among several processes, we could see that it was possible to avoid such performance penalties by designing the applications so that they explicitly managed this synchronization. Our conclusion is that a programmer is generally much better off using the facility than building a custom one for generic applications.

1. INTRODUCTION

Remote procedure calls have been employed in network systems for several years, e.g., see [11], and continue to be refined by today's researchers, e.g, see [2, 3, 9]. Other researchers are extending the basic remote procedure call model to satisfy other requirements such as bulk data transfer, e.g., see Gifford and Glasser's channel model [4], and distributed object-oriented system support, e.g. see Bennett's distributed Smalltalk [1] and Jul et al's Emerald system [6]. There are also commercial products that provide remote procedure call facilities for application programming, e.g., Sun [10] and Xerox [12]; IBM also employs remote procedure call techniques in the SNA LU 6.2 protocol [5].

While remote procedure call mechanisms do not provide for simultaneous operation between distinct portions of a program, they do allow a programmer to cause procedures to be executed in different environments. Such an application promotes sharing of information among the processes using a familiar model of computation -- the calling program views remote execution as a conventional procedure call.

For example, remote procedure calls can be used to check the status of a remote machine environment. The Sun documentation uses an example in which a remote procedure, **rnusers()**, can check its host machine to determine the number of users currently logged into the machine [10]. The procedure can be called remotely so that a local program can check the number of users on various machines in a network by remotely calling **rnusers()**; it is not necessary to log into each machine to run a local status program. This program is typical of simple procedures used to check the status of some remote machine, e.g., disk availability or load statistics.

Another class of applications are those that not only check status, but also change information in the remote machine. For example, the Sun documentation also describes remote procedures that update the yellow pages server from a client machine. Of course, remote procedures are natural for implementing arbitrary (atomic) transactions on servers, provided that the procedure server is constructed to serialize such transaction executions.

The stimulation for the performance work described in this paper came from our consideration of a remote procedure call facility as a program interpretation facility. In some of our research we rely on a graph interpretation server to be able to execute node interpretations that are defined in an arbitrary language, yet to do so "on demand" of the server (which is driven by the semantics of the graph models), see below. We needed a facility that would allow us to separate the language of graph interpretations from the server implementation, and which would provide us with a late-binding mechanism. The remote procedure call facility provides that capability, since remote procedures are separately compiled and linked through an interface that allowed for flexible binding of calling and called procedures.* We observed that if remote procedure call could be used in our application, then we would be able to distribute graph node interpretation away from the machine that managed the parallelism in the graph operation. We could then approximate the parallel operation of the graph through distributed interpretation of individual nodes. Our hesitation in using rpc stemmed only from performance concerns: Would remote procedure calls be too costly for our application?

Remote procedure call facilities are briefly described below; we also describe our application of remote procedure calls. Next, we describe our measurements of Sun's implementation of remote procedure calls in an attempt to gather quantitative information to decide upon the utility of the approach for our particular problem.

2. REMOTE PROCEDURE CALL IMPLEMENTATION

2.1. Conceptual Remote Procedure Call Operation

Briefly, a remote procedure call (*rpc*) facility is intended to allow a programmer to write a pair of programs, one a calling program and the other a procedure, such that each is executed in a distinct environment, e.g., on distinct machines.

An rpc facility is implemented from five components (see Figure 1): A *user*, a *local-stub*, a *network*, a *remote-stub*, and the *remote procedure* [3,4].

The machine that supports the user links the remote procedure call to a local routine in the local-stub. The local-stub encodes the procedure name (previously registered with the remote-stub), packages the parameters, and communicates the information to the remote-stub (a surrogate main program) on the remote server machine. The library code causes the calling process to block until a response is received from the remote machine.

---

* Of course some programming environments provide a facility for late binding of procedures. However, we were constrained to implement our solution in the Sun C/Unix environment since our graph interpreter is implemented in that environment.

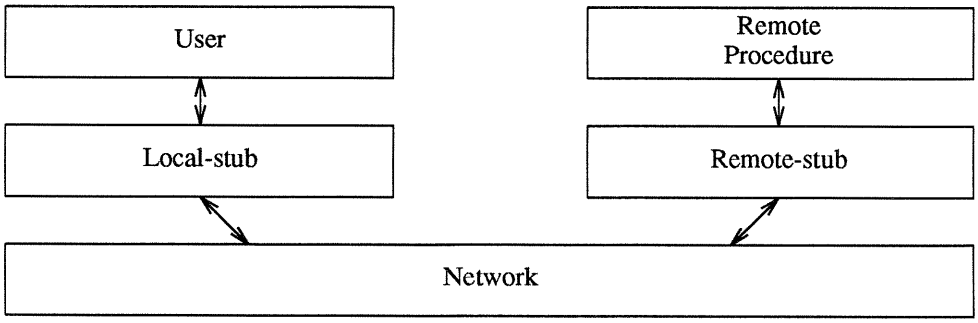

Figure 1: A Schematic of Remote Procedure Call

The remote procedure server machine executes the remote-stub, which solicits requests from the interprocess communication facility. When a procedure invocation request arrives, the remote-stub decodes the request using the procedure registry table, unpacks the parameters, then invokes the remote procedure. When the remote procedure returns to the remote-stub, the results are packaged and returned to the local-stub via the interprocess communication mechanism.

The local-stub in the client process accepts the returned results from the remote-stub, and uses conventional local language mechanisms to return the values to the calling program.

There are many details omitted from this description, e.g., how the local-stub locates the remote-stub, how the remote procedure is bound to a server, etc. We refer the interested reader to one of the discussions about rpc implementations for additional detail.

2.2. The Sun Remote Procedure Call Facility

The Sun Remote Procedure Call (*RPC*) library (see "Remote Procedure Call Programming Guide" in [10]), is a production-level implementation of the rpc facility described above.

By its use of the External Data Representation (XDR) library, (see "External Data Representation Protocol Specification" in [10]), it also allows the arguments and results for such calls to be compatible between machines with different word sizes and data alignment conventions.

Figure 2 identifies the protocol stack used in the Sun RPC implementation. There are three different interfaces to the RPC facility in the Sun networking package. The highest layer provides transparent use of the mechanism to call a predefined set of routines -- RPC Service Library Routines. The medium level interface provides an interface to the registry operation and makes the remote call explicit; to use the medium level, the calling program must be able to locate the server that contains the desired procedure. The lowest level interface provides maximum flexibility in specifying attributes of the interprocess communication mechanism, etc. For example, the medium level interface employs UDP for the call/return communication. If the user desires to send more than 8 KB of data to the procedure, then it will be necessary to employ TCP; this can only be done by using the lowest level RPC interface.

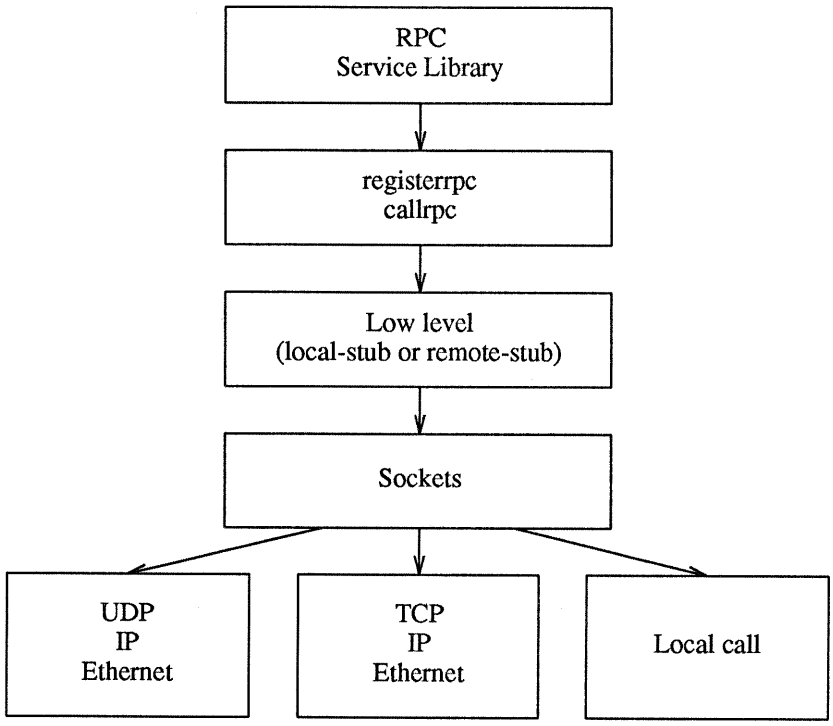

Figure 2: Sun Remote Procedure Call Implementation

The lowest level of the RPC package is implemented on top of sockets [7]. Sockets, in turn, can be used for local IPC, or they can use various network protocol implementations to accomplish IPC among processes on different machines.

Default initialization of a socket causes it to be constructed with a blocking read interface. The socket library also includes a *select* call to simplify the usage of blocking read in certain cases: if the user calls *select* on a socket, then the call will return immediately if the socket contains data that can be read, or will return after a specified amount of time (the *timeout* interval with a result that indicates that there is no data to be read. The default RPC usage of sockets incorporates calls to *select*.

Both TCP and UDP protocol layers are implemented on IP which, in our installation, is implemented on Ethernet.

2.3. Applying RPC to the Graph Interpretation Problem

The Olympus Modeling System is composed of a frontend and a backend [8]. The frontend is a graphics editor and console to control precedence graph interpretation in the backend. Graphs are marked with tokens, as in Petri nets; a node in the graph fires whenever the configuration of tokens associated with the node satisfy the node's firing rules. Each node can be annotated with a procedure to describe the action that the node represents in the model. Thus, from the point of view of the Olympus backend, it is required that the procedure associated with a node be interpreted whenever the node is fired.

Olympus is intended to support many variants of this scenario, each variant having specific firing rules and specific languages in which the node interpretation is expressed. As designers of Olympus, our requirement was to provide interpretation facilities for the node interpretation languages.

There are several acceptable languages supported in the general RPC environment provided by Sun, including C. The RPC facility is language-independent, provided that the interface is used properly. Our modeling work allowed us to utilize C as the node interpretation language, (although future work will undoubtedly focus on languages such

as Prolog). It is a straight-forward application to modify the Olympus backend so that the node firing implementation looks up the name of a remote procedure and uses RPC to call it.

By the nature of our class of graph models, there is considerable parallelism represented in the model. Because of our desire to run the interpretation in scaled real time, and our desire to run it as fast as possible at other times, it is desirable to attempt to execute the node interpretations simultaneously if possible. While it is not the focus of this paper to discuss this aspect of the work, it is noteworthy that we decided that we could take advantage of the RPC mechanism to implement such an approach by extending the lowest level of the RPC mechanism to support a *non-blocking remote procedure call* with an appropriate return mechanism.

It is clear that there may be severe performance penalties from using RPC to support node interpretation in Olympus, although the use of RPC could save considerable time in implementing our system.

2.4. Performance Considerations

In his early paper, White pointed out that rpc is much more time-consuming than local procedure calls [11]. He urged that the user consider the time tradeoffs before embracing the technique. However, his early work did not quantify the cost of employing the technique.

Any given rpc implementation is built on several layers of abstract machine, and is susceptible to performance degradation due to inefficient implementation. Since the implementations are complex, it also seems conceivable that a given implementation may perform well under one usage pattern, yet be inadequate for another.

A search into the literature revealed that performance was a recurring worry for every rpc implementation. For example, Birrell and Nelson clearly indicate that they had a performance goal such that their implementation of rpc should not exceed the time for network transmission time by a factor of five [3]. Their measurement results were specific to the XNS implementation, and did not apply to the Sun RPC mechanism.

Researchers on the HCS project describe an rpc facility (HRPC) for heterogeneous networks [2]. Here, the facility must be sufficiently general to span any particular rpc implementation. The performance data from the study were used to illustrate that the HRPC facility performed similarly to the Sun RPC and the Courier RPC. While the paper illustrates that HRPC is competitive with the commercial offerings, it did not really provide us with the detailed performance information we required in order to proceed with our application.

Gifford and Glasser also collect performance information on an rpc implementation [4]. However, the implementation is their own channel model implementation (generalized to handle bulk data transfer). The channel model is built on top of 4.2BSD TCP, and yielded acceptable performance, so our application still seemed plausible, even though we were unable to find performance data about the Sun RPC in the literature.

While it appeared that some rpc approach was feasible, we wondered if Sun's RPC and XDR would degrade performance over a custom rpc mechanism. The argument for a custom rpc would be that unneeded features, perhaps such as XDR, could be eliminated resulting in better performance.

Since the implementation of such facilities involves significant design, coding, and testing, timing results on the libraries are desirable for providing a good measure of what areas can be improved, and how much improvement can be expected. Specifically, with a knowledge of the networking facilities provided by sockets, one could attempt to increase performance along at least three dimensions:

(1) Procedure dispatching. While this is not likely to be a problem, it might be argued that in a particular application program, one does not need the full generality of the naming conventions that are used to make RPC calls. Also, if this is a significant cost, clever methods of storing the dispatching information to facilitate fast lookup are widely known and could be implemented.

(2) Data format independence. If an application is known to be targeted only to a single machine or a network of identical machines, then it could run without using the XDR protocols.

(3) Error detection and recovery, and timeouts. The Sun libraries provide detailed error information on both the local and remote sides of the interface, and allow the specification of a maximum time to wait for a reply. If these facilities were deemed less important than speed, they might be sacrificed in a custom implementation. Also, particularly in a research setting, one might be able to guarantee to some level of certainty that a server process is always running and will always return a result properly when called. Thus, the timeout facility would not be needed and could be eliminated if it turned out to be costly.

The conclusion was that it was worthwhile to conduct some experiments on RPC to supplement the information that we had found in the literature, and which would be particular to our Sun RPC environment.

3. THE TIMING EXPERIMENTS

3.1. Experiment Description

With the aim of gaining insight into the details of RPC, four different programs were studied on a Sun 3/50, operating as a client served by NFS. A brief description of each program follows.

Program One

To provide a base line for interpreting other results, this program measures the cost of a normal C function call. The program is instrumented to reflect the time required for executing an empty C for-loop, a loop with a call with no arguments, a 2-byte argument, or a four-byte argument. The empty loop experiment is used to calibrate the timing information since the granularity of the system time function is only 16.67 msec for a 3/50.

Program Two

This program uses the medium level of the RPC mechanism to call three remote procedures, both on the local machine and a remote machine across the network, i.e., this experiment employs the default settings on sockets such as UDP support with default settings on the retry attempts on call and default timeout on *select* calls. These procedures differ only in that they take either no arguments, a 2-byte argument, or a 4-byte argument. All three procedures return no result.

Program Three

This is a custom program that uses its own primitive (and therefore presumably efficient) RPC protocol, implemented on top of UDP sockets. Since the basic RPC mechanism also uses datagram sockets, this program should be comparable to Program Two; the program is intended to be used to determine the relative cost of communications and XDR compared to the cost of the remote procedure calls themselves. In particular, the dispatching mechanism on the server side is simply a C switch-statement. On the client side the calling information is not kept in a table, but rather is hard-coded into building the remote call message buffer.

Two versions of this program are provided: one makes the remote call and then blocks waiting for the reply; the other uses the *select* system call to determine when and if a reply is received. The three procedures each return an integer result to indicate that the remote call has completed.

Program Four

This is identical to Program Three, but uses stream (TCP) sockets.

3.2. The Validity of the Measurements

The timing data are obtained using the *clock* library function, calling it before and after each loop and subtracting the two results to get the elapsed time. Since the clock granularity is approximately 16667 μsec and 1000 iterations were executed, the expected uncertainty due to the clock would be approximately ±33 μsec. However, repeatedly running the no-argument local RPC calls yields the following results:

Number of Occurrences	Time/iteration (μsec)
1	3967
1	4000
1	4050
3	4100
2	4133
2	4150
3	4167
1	4183
1	4200
1	4216
1	4233
3	4250
3	4283
2	4383
average = 4183 ±200	

Table 1

This indicates that the actual uncertainty is greater than expected, probably due to system activity variations. We can expect similar uncertainty on remote calls due to network traffic. These results should be considered when analyzing the measured times; however, even this greater variation only amounts to about 5%, so we can still draw meaningful conclusions from the experimental data. Finally, it should be noted that the timing data and the above statistical data were obtained at approximately the same time, on the same system, with the same types and levels of user activity.

3.3. RPC Measurement Results

The measurement results are summarized in the following tables. Table 2 shows the base line cost of local procedure calls. The time/iteration for the empty loop is approximately 1.2 μsec. When the procedure call is introduced, approximately 4.1 μsec are required for the case with no arguments, 5.4 μsec for the 2-byte parameter case and 4.8 μsec for the 4-byte parameter case.

C Function Calls				
Iterations	Time/iteration (μsec)			
	Empty loop	No arg	2 byte	4 byte
100000	1.333280	4.166500	5.666440	4.999800
1000000	1.233284	4.133168	5.449782	4.866472
10000000	1.213285	4.061504	5.473114	4.893138

Table 2

Table 3 shows the performance for the three programs that use RPC (based on a 1000 iteration loop for clock resolution). First, it is evident that even for a local call, the RPC mechanism introduces an overhead of about three orders of magnitude, i.e., 4.2 msec versus 4.1 μsec in the raw C case (for no arguments). Similar differences are apparent in the 2- and 4-byte cases.

Remote Procedure Calls - 1000 Iterations										
Machine	Block/ Select	Time/iteration (µsec) - No arg, 2 byte, 4 byte								
		Program 3 (UDP Sockets)			Program 2 (RPC)			Program 4 (TCP Sockets)		
Local	Block	2950	2883	2783	4216	3950	4183	3517	3017	3550
	Select	3033	3033	3150				4150	4583	4317
Remote	Block	3900	3767	3783	5150	5283	5266	3400	3333	3517
	Select	4483	4050	4750				5083	4666	5116

Table 3

Comparing the cost of local and remote use of RPC, we see that the performance drops off about 25% for remote calls versus local calls. The percentage differences were larger, up to 50%, for the custom implementation that uses *select* with UDP (Program Three), but smaller, less than 23%, for all cases on TCP. Based on these observations, the idea of implementing the procedures remotely versus local implementation did not seem critical. Also, the cost of using the mechanism in this environment is clear.

Is the custom implementation (Program Three) significantly better than the standard RPC (Program Two), i.e., what is the cost of XDR? While we observed about a 20% cost between the version of Program Three that uses *select* and Program Two, it is difficult to conclude that this difference is only attributable to the use of XDR. It is also apparent that the timeout parameters on the select call were having a significant influence on performance (by comparing the *select* version of Program Three with the blocking read version).

In order to gain a better understanding of the cause of our observations, we decided to profile some of the programs.

3.4. Profiling Data

In order to shed more light on the differences in timing noted between the versions of Programs Three and Four that blocked or used *select*, the programs were profiled yielding the following information (only data for the most significant amounts of time are shown).

Program 3 (UDP Sockets) - 1000 Iterations				
%time	cumsecs	#call	ms/call	name
Blocking				
69.4	5.16	3004	1.72	_sendto
28.0	7.24	3003	0.69	_recvfrom
1.1	7.32			mcount
0.8	7.38	1	60.00	_main
0.3	7.40	2	10.00	_sbrk
0.3	7.42	3	6.67	_select
0.3	7.44	7	2.86	_xdrmem_create

Table 4a

Program 3 (UDP Sockets) - 1000 Iterations				
%time	cumsecs	#call	ms/call	name
Using Select				
58.9	4.90	3004	1.63	_sendto
24.3	6.92	3003	0.67	_select
14.4	8.12	3003	0.40	_recvfrom
1.0	8.20			mcount
0.7	8.26	1	60.00	_main
0.2	8.28			_clntudp_create
0.2	8.30	1	20.00	_fstat
0.2	8.32	1	20.00	_write

Table 4b

This can be compared with the same profiling of the RPC code.

Program 2 (RPC) - 1000 Iterations				
%time	cumsecs	#call	ms/call	name
40.5	5.34	3004	1.78	_sendto
16.5	7.52	3004	0.73	_recvfrom
15.8	9.60	3004	0.69	_select
7.0	10.52			mcount
3.9	11.04	3010	0.17	_xdrmem_create
3.3	11.48	1	439.98	_clntudp_create
2.1	11.76	13024	0.02	_xdr_long
1.5	11.96	3000	0.07	_callrpc
1.4	12.14	1	179.99	_main
1.1	12.29	3004	0.05	_xdr_accepted_reply
1.1	12.43	12024	0.01	_xdr_enum
0.9	12.55	3017	0.04	_bcopy
0.9	12.67	3000	0.04	_strcmp
0.6	12.75	3009	0.03	_xdr_bytes
0.6	12.83	3006	0.03	_xdr_opaque_auth
0.6	12.91	3005	0.03	_xdr_union
0.5	12.97	3004	0.02	__seterr_reply
0.3	13.01	5	8.00	_authnone_create
0.3	13.05	3004	0.01	_xdr_replymsg
0.3	13.09	6047	0.01	_xdr_u_long
0.3	13.13	4000	0.01	_xdr_void
0.2	13.15	2	10.00	_getsockname
0.2	13.17	7	2.86	_ioctl
0.2	13.19	1000	0.02	_xdr_short
0.1	13.20			_xdr_rejected_reply

Table 5

At this point we mention only that it is clear the call to *select* is somewhat expensive. The *select* call appears to add approximately 0.88 sec to the total running time of the program. Averaging this on a per-loop basis gives 0.29 sec additional time per loop, or 290 μsec per iteration. This is an overestimate of course, since it averages over the whole program, and we can see that 3 additional calls to *select* are made outside the 3 loops of 1000 iterations each. (More in-depth analysis reveals that these are produced by NFS.) Looking back at the timing measurements, we see that the average difference between blocking and using *select* for local calls in Program Three is 200 μsec per iteration. Thus, the timing data roughly agree with the information provided by profiling. The implications of the profiling information for program two (using Sun RPC) relative to the timing data are discussed below.

4. CONCLUSIONS

By eliminating enough of the RPC functionality, we were able on average to improve performance about 29%. To achieve this, we specifically:

(1) Simplified procedure dispatching in both the client and the server by hard-coding the calling information and using a switch-statement for dispatching in the server. As anticipated, this is does not appear to be a significant contributor to the cost of remote calls, judging from the profiling information.

(2) Eliminated the use of XDR routines for providing machine-independent data format. This will be examined further below.

(3) Removed the ability to receive replies from several servers and to time out in the event of some error, by just blocking until a reply to a remote call is received.

The performance improvement drops to an average of about 20% when we change (3) above by using the *select* system call instead of blocking on replies. This still does not provide robust error reporting or handling, since it only provides for multiplexing return messages and timing out if either the called host or the network is out of service.

We hypothesize that the remaining cost is due to some combination of procedure dispatching and the overhead of using XDR. To examine this further, we look closely at the profiling data for Program Two, which did local calls using RPC.

In the profiling data for this program, if we look at the routines which were called more than 1000 times (which were calls made inside the loops, and therefore represent the cost of the calls themselves) we can easily see which routines are associated with XDR. The only one that is not obvious is *bcopy*, which more detailed profiling (using the *gprof* utility) reveals is called from the *xdrmem_create* routine, and therefore should be included in the cost of using XDR. Together, these 13 routines account for 1644 msec total, or 584 μsec per iteration. It should be noted that this actually subsumes several of the costs mentioned above: it reflects the overhead of using XDR, and also includes the cost of communicating the additional data that RPC uses in order to support the [program, version, procedure] identification of remote procedures and the reporting of errors on the server side.

The cost of dispatching on the client side can be found by looking at the 2 routines (*callrpc* and *strcmp*) that are associated with dispatching the calls. (Again, detailed profiling reveals that *strcmp* is called from *callrpc*.) These account for 330 msec total, or 110 μsec per iteration.

Thus, it appears that XDR accounts for an average of 14% of the cost per iteration in Program Two, while dispatching accounts for about 3%.

Given the uncertainty discussed above, we may conclude that this total of about 700 μsec per iteration (17%) accounts for the remaining difference between Program Two and Program Three using the *select* call, since Program Three does not use XDR at all and also does not incur any significant cost in dispatching since it does no work looking up procedures to make the calls.

These results indicate that one can improve remote procedure call performance. However, the simplifications that must be made are in general not desirable. The elimination of *select*, which is a significant factor, cannot be done unless the application can tolerate blocking while waiting for results. This could result in deadlock if the remote host goes down, the remote procedure contains an error and fails to return, or the network suffers a failure. The dispatching cost, while small, could be attacked. RPC keeps its registered procedures on a linked list, so that dispatching cost will increase linearly with the number of procedures. This will only become a factor, however, in the case of a very large number of remotely called procedures. Thus, working here to increase performance is probably not an economical use of programmer effort. Finally, XDR is a significant contributor to the cost of RPC. While one might not presently anticipate the need for machine-independence, it is a rule of software engineering that such assumptions usually are contradicted as soon as considerable effort is invested on the basis of them. It should also be considered that XDR provides some very useful memory allocation facilities in addition to a portable data representation.

In total, the data seem to indicate that it is not worth the effort to design and implement a custom version of RPC unless one has an application that is very simple, does not need to be very robust, and needs to run very fast. It also must be kept in mind that there is a bound on the improvement that can be made, and that the largest part of the cost, about 80%, is in the communication using sockets. Since a thousand-fold increase in performance can be made by not using remote calls at all, a distributed program's efficiency is more likely to be greatly improved not by attempting fine-tuning on the RPC mechanism, but rather by using the existing mechanism in a well thought-out manner.

While we have accepted the fact that remote procedures are generally expensive, and that we are not likely to speedup our overall graph interpretation facility through their use, it has proven to be extremely useful for the ease with which we can provide interpretations to our graph models.

The current version of Olympus uses the nonblocking interface to the lowest level RPC facility, allowing us to rapidly proceed with our modeling ideas at the identified cost of runtime performance.

5. ACKNOWLEDGEMENTS

This research has been supported by NSF cooperative agreement DCR-8420944, NSF Grant No. CCR-8802283, and U S West Advanced Technologies.

6. REFERENCES

1. J. K. Bennett, "Distributed Smalltalk: Inheritance and Reactiveness in Distributed Systems", Department of Computer Science, University of Washington, Ph.D dissertation, December 1987.

2. B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz, "A Remote Procedure Call Facility for Heterogeneous Computer Systems", *IEEE Transactions on Software Engineering SE-13*, 8 (August

1987), 880-894.

3. A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59.
4. D. K. Gifford and N. Glasser, "Remote Pipes and Procedures for Efficient Distributed Communication", *ACM Transactions on Computer Systems 6*, 3 (August 1988), 258-283.
5. *An Introduction to Advanced Program-to-Program Communication (APPC)*, IBM International Systems Center, IBM Corporation, Technical Bulletin GG24-1584-0, July 1983.
6. E. Jul, H. Levy, N. Hutchinson and A. Black, "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109-133.
7. S. J. Leffler, R. S. Fabry, W. N. Joy and P. Lapsley, "An Advanced 4.3BSD Interprocess Communication Tutorial", in *Unix Programmer's Manual Supplementary Documents 1*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.
8. G. J. Nutt, "Olympus: An Extensible Modeling and Programming System", Technical Report No. CU-CS-412-88, Department of Computer Science - University of Colorado, Boulder, October 1988.
9. M. F. Pucci and J. L. Alberi, *Genralized Interprocessor Communication in the Architecture of the Dune Distributed Operating System*, Bell Communications Research Technical Memorandum TM-ARH-010642, 1987.
10. "Networking on the Sun Workstation", Document Number 800-1345-10, Sun Microsystems, Inc., September 1986.
11. J. White, "A High-Level Framework for Network-Based Resource Sharing", *Proceedings of the National Computer Conference*, 1976, 561-570.
12. *Courier: The Remote Procedure Call Protocol*, Xerox System Integration Standard XSIS 038112, Xerox Corporation, December 1981.