Performance of Firefly RPC

MICHAEL D. SCHROEDER and MICHAEL BURROWS

April 15, 1989

SRC RESEARCH REPORT 43

AUTHORS' ABSTRACT

In this paper, we report on the performance of the remote procedure call implementation for the Firefly multiprocessor and analyze the implemen-tation to account precisely for all measured latency. From the analysis and measurements, we estimate how much faster RPC could be if certain improve-ments were made.

The elapsed time for an inter-machine call to a remote procedure that accepts no arguments and produces no results is 2.66 milliseconds. The elapsed time for an RPC that has a single 1440-byte result (the maximum result that will fit in a single packet) is 6.35 milliseconds. Maximum inter-machine throughput using RPC is 4.65 megabits/second, achieved with 4 threads making parallel RPCs that return the maximum sized single packet result. CPU utilization at maximum throughput is about 1.2 on the calling machine and a little less on the server.

These measurements are for RPCs from user space on one machine to user space on another, using the installed system and a 10 megabit/second Ethernet. The RPC packet exchange protocol is built on IP/UDP, and the times include calculating and verifying UDP checksums. The Fireflies used in the tests had 5 MicroVAX II processors and a DEQNA Ethernet controller.

TABLE OF CONTENTS

1. INTRODUCTION

Remote procedure call (RPC) is now a widely accepted method for encapsulating communication in a distributed system. With RPC, programmers of distributed applications need not concern themselves with the details of managing communications with another address space or another machine, nor with the detailed representation of operations and data items on the communication channel in use. RPC makes the communication with a remote environment look like a local procedure call.

In building a new software system for the Firefly multiprocessor [9] we decided to make RPC the primary communication paradigm, to be used by all future programs needing to communicate with another address space, whether on the same machine or a different one. Remote file transfers as well as calls to local operating systems entry points are handled via RPC. For RPC to succeed in this primary role it must be fast enough that programmers are not tempted to design their own special purpose communication protocols. Because of the primary role of RPC, however, we were able to structure the system software to expedite the handling of RPCs and to pay special attention to each instruction on the RPC "fast path".

This paper reports measurements of Firefly RPC performance. It also details the steps of the fast path and assigns an elapsed time to each step. Correspondence of the sum of these step times with the measured overall performance indicates that we have an accurate model of where the time is spent for RPC. In addition, this detailed understanding allows estimates to be made for the performance improvements that would result from certain changes to hardware and software.

1.1 Hardware and System Characteristics

The Firefly multiprocessor allows multiple VAX processors access to a shared memory system via coherent caches. The Firefly version measured here had 16 megabytes of memory and 5 MicroVAX II CPUs [9], each of which provides about 1 MIPs of processor power[1]. One of these processors is also attached to a QBus I/O bus [5]. Network access is via a DEQNA device controller [4] connecting the QBus to a 10 megabit/second Ethernet. In the Firefly the DEQNA can use about 16 megabits/second of QBus bandwidth.

The Firefly system kernel, called the Nub, implements a scheduler, a virtual memory manager, and device drivers. The Nub executes in VAX kernel mode. The virtual memory manager provides multiple user address spaces for application programs, one of which contains the rest of the operating system. The scheduler provides multiple threads per address space, so that the Nub, operating system, and application programs can be written as true concurrent programs that execute simultaneously on multiple processors. The system is structured to operate best with multiple processors.

1.2 Overview of RPC Structure

The Firefly RPC implementation follows the standard practice of using stub procedures [2]. The caller stub, automatically generated from a Modula-2+ [8] interface definition, is included in the calling program to provide local surrogates for the actual remote procedures. When a procedure in this stub is called, it allocates and prepares a call packet into which are marshalled the interface and procedure identification, and the arguments. The stub calls the appropriate transport mechanism to send the call packet to the remote server machine and then blocks, waiting for a corresponding result packet. (Other threads

---

[1] Since the measurements reported here were made, Fireflies have been upgraded with faster CVAX processors and more memory .

in the caller address space are still able to execute.) When the result packet arrives, the stub unmarshalls any results, frees the packet, and returns control to the calling program, just as though the call had taken place within the same address space.

Similar machinery operates on the server. A server stub is included in the server program. This stub receives calls from the transport mechanism on the server machine when a suitable call packet arrives. The stub unmarshalls the arguments and calls the identified procedure. After completing its task, the server procedure returns to the stub, which marshalls the results and then calls the appropriate transport mechanism to send the result packet back to the caller machine.

More details on the structure of Firefly RPC appear in section 3.

2. MEASUREMENTS

In this section we report the overall performance of Firefly RPC. All measurements in this paper were made on the installed service system, software that was used by more than 50 researchers. Except where noted all tests used automatically generated stubs for a remote "Test" interface that exports three procedures:

```
PROCEDURE: Null();
PROCEDURE: MaxResult(VAR OUT buffer: ARRAY OF CHAR);
PROCEDURE: MaxArg(VAR IN buffer: ARRAY OF CHAR);
```

MaxArg and MaxResult were called with the following variable as the argument:

```
VAR b: ARRAY [0..1439] OF CHAR;
```

Calls to Null() measure the base latency of the RPC mechanism. The Ethernet packets generated for the call and return of this procedure, which accepts no argument and produces no result, consist entirely of Ethernet, IP, UDP, and RPC headers and are the 74-byte minimum size generated for Ethernet RPC.

Calls to MaxResult(b) measure the server-to-caller throughput of RPC. The single 1440-byte VAR OUT argument produces the minimal 74-byte call packet and a result packet with 1514 bytes, the maximum allowed on an Ethernet. (The RPC implementation allows arguments and results larger than 1440 bytes, but such larger arguments and results necessarily are transmitted in multiple packets.) The VAR OUT designation tells the RPC implementation that the argument value need only be transferred in the result packet. MaxArg(b) moves data from caller to server in the same way. The VAR IN designation means that the argument value need only be transferred in the call packet.

2.1 Latency and Throughput

As an overall assessment of RPC performance on the Firefly, we measured the elapsed time required to make a total of 10000 RPCs using various numbers of caller threads. The caller threads ran in a user address space on one Firefly, and the multithreaded server ran in a user address space on another. Timings were done with the two Fireflies attached to a private Ethernet to eliminate variance due to other network traffic.

Table I: Time for 10000 RPCs

# of caller threads	Calls to Null()		Calls to MaxResult(b)	
	seconds	RPCs/sec	seconds	megabits/sec
1	26.61	375	63.47	1.82
2	16.80	595	35.28	3.28
3	16.26	615	27.28	4.25
4	15.45	647	24.93	4.65
5	15.11	662	24.69	4.69
6	14.69	680	24.65	4.70
7	13.49	741	24.72	4.69
8	13.67	732	24.68	4.69

From Table I we see that the base latency of the Firefly RPC mechanism is about 2.66 milliseconds and that 7 threads can do about 740 calls of Null() per second. Latency for a call to MaxResult(b) is about 6.35 milliseconds and 4 threads can achieve a server-to-caller throughput of 4.65 megabits/second using this procedure. We observed about 1.2 CPUs being used on the caller machine, slightly less on the server machine, to achieve maximum throughput. Those Fireflies, which had all the standard background threads started, used about 0.15 CPUs when idling.

2.2 Marshalling Time

RPC stubs are automatically generated from a Modula-2+ definition module. The stubs are generated as Modula-2+ source, which is compiled by the normal compiler. For most argument and result types, the stub contains direct assignment statements to copy the argument or result to/from the call or result packet. Some complex types are marshalled by calling library marshalling procedures.

Andrew Birrell has measured the following times for passing various argument types with the automatically generated stubs. The measurements reported are the incremental elapsed time for calling a procedure with the indicated arguments over calling Null(). The differences were measured for calls to another address space on the same machine in order to factor out the Ethernet transmission time for different sizes of call and result packets. Such local RPC[1] uses the same stubs as inter-machine RPC. Only the transport mechanism is different: shared memory rather than IP/UDP and the Ethernet. Because the pool of packet buffers (the same pool used for Ethernet transport) is mapped into each user address space, the time for local transport is independent of packet size.

Table II: 4-byte integer arguments, passed by value

# of arguments	Marshalling time in microseconds
1	8
2	16
4	32

Integer and other fixed-size arguments passed by value are copied from the caller's stack into the call packet by the caller stub, and then copied from the packet to the server's stack by the server stub. Such arguments are not included in the result packet.

---

[1] The time for a local RPC to Null() is 937 microseconds for the system measured here. See Bershad et al. [1] for a report on successful efforts to speed up local RPC on the Firefly.

Table III: Fixed length array, passed by VAR OUT

Array size in bytes	Marshalling time in microseconds
4	20
400	140

Table IV: Variable length array, passed by VAR OUT

Array size in bytes	Marshalling time in microseconds
1	115
1440	550

In Modula-2+, VAR arguments are passed by address. The additional OUT or IN designation tells the stub compiler that the argument is being passed in one direction only. The stub can use this information to avoid transporting and copying the argument twice. A VAR OUT argument is transported only in the result packet; it is not copied into the call packet by the caller stub. If the argument fits in a single packet then the server stub passes the argument's address in the result packet buffer to the server procedure, from where the server procedure can directly write it, so no copy is performed at the server. The single copy occurs upon return when the caller stub moves the value in the result packet back into the caller's argument variable. VAR IN arguments work the same way, *mutatis mutandis*, to transfer data from caller to server. VAR OUT and VAR IN arguments of the same type have the same incremental marshalling costs. For single packet calls and results the marshalling times for array arguments scale linearly with the values reported in tables III and IV.

Table V: Text.T argument

Array size in bytes	Marshalling time in microseconds
NIL	89
1	378
128	659

A Text.T is a text string that is allocated in garbage collected storage and is immutable. The caller stub must copy the string into the call packet. The server stub must allocate a new Text.T from garbage collected storage at the server, copy the string into it, and then pass a reference to this new object to the server procedure. Most of the time for marshalling Text.Ts is spent in the Text library procedures.

3. ANALYSIS

In this section we account for the elapsed time measured in section 2.1. We start by describing in some detail the steps in doing an inter-machine RPC. Then we report the time each step takes and compare the total for the steps to the measured performance.

3.1 Steps in a Remote Procedure Call

The description here corresponds to the fast path of RPC. It assumes that other calls from this caller address space to the same remote server address space have occurred recently, within a few seconds, so that server threads are waiting for the call. Part of

making RPC fast is arranging that the machinery for retransmission, for having enough server threads waiting, for multi-packet calls or results, for acknowledgements, and other features of the complete RPC mechanism intrude very little on the fast path. Consequently, the description of the fast path can ignore these mechanisms. The path described is that followed for the majority of RPCs that occur in the operational system. It is this fast path that determines the normally observed performance of RPC.

Firefly RPC allows choosing from several different transport mechanisms at RPC bind time. Our system currently supports transport to another machine by a custom RPC packet exchange protocol layered on IP/UDP, by DECNet to another machine, and by shared memory to another address space on the same machine. The choice of transport mechanism is embodied in the particular versions of the transport procedures named Starter, Transporter, and Ender that are invoked by the caller stub. At the server the choice is represented by the Receiver procedure being used. In this paper we measure and describe the first of these transport options, using Ethernet. This custom RPC packet exchange protocol follows closely the design described by Birrell and Nelson for Cedar RPC [2]. The protocol uses implicit acknowledgements in the fast path cases.

3.1.1 Caller stub

When a program calls a procedure in a remote interface, control transfers to a caller stub module for that interface in the caller's address space. Assuming that binding to a suitable remote instance of the interface has already occurred, the stub module completes the RPC in five steps:

1. Call the Starter procedure to obtain a packet buffer for the call with a partially filled-in header.
2. Marshall the caller's arguments by copying them into the call packet.
3. Call the Transporter procedure to transmit the call packet and wait for the corresponding result packet.
4. Unmarshall the result packet by copying packet data to the caller's result variables.
5. Call the Ender procedure to return the result packet to the free pool.

When the stub returns control to the calling program, the results are available as if the call had been to a local procedure.

3.1.2 Server stub

The server stub has a similar job to do. When it receives a call packet on an up call from the Receiver procedure on the server machine, it performs three steps:

1. Unmarshall the call's arguments from the call packet. Depending on its type, an argument may be copied into a local stack variable, copied into newly allocated garbage collected storage, or left in the packet and its address passed. The call packet is not freed.
2. Call the server procedure.
3. When the server procedure returns, marshall the results in the saved call packet, which becomes the result packet.

When the server stub returns to the Receiver procedure, the result packet is transmitted back to the caller.

3.1.3 Transport mechanism

The Transporter procedure must fill in the RPC header in the call packet. It then calls the Sender procedure to fill in the UDP, IP, and Ethernet headers, including the UDP checksum on the packet contents. To queue the call packet for transmission to the server machine, the Sender invokes the Ethernet driver, by trapping to the Nub in kernel mode, .

Because the Firefly is a multiprocessor with only CPU 0 connected to the I/O bus, the Ethernet driver must run on CPU 0 when notifying the Ethernet controller hardware. Control gets to CPU 0 through an interprocessor interrupt; the CPU 0 interrupt routine prods the controller into action.

Immediately after issuing the interprocessor interrupt, the caller thread returns to the caller's address space where the Transporter registers the outstanding call in an RPC call table, and then waits on a condition variable for the result packet to arrive. The time for these steps is not part of the fast path latency as the steps are overlapped with the transmission of the call packet, the processing at the server, and the transmission of the result packet. For the RPC fast path the calling thread gets the call registered before the result packet arrives.

Once prodded, the Ethernet controller reads the packet from memory over the QBus and then transmits it to the controller on the server machine. After receiving the entire packet, the server controller writes the packet to memory over the server QBus and then issues a packet arrival interrupt.

The Ethernet interrupt routine validates the various headers in the received packet, verifies the UDP checksum, and then attempts to hand the packet directly to a waiting server thread. Such server threads are registered in the call table of the server machine. If the interrupt routine can find a server thread associated with this caller address space and called address space, it attaches the buffer containing the call packet to the call table entry and awakens the server thread directly.

The server thread awakens in the server's Receiver procedure.[1] The Receiver inspects the RPC header and then calls the the stub for the interface ID specified in the call packet. The interface stub then calls the specific procedure stub for the procedure ID specified in the call packet.

The transport of the result packet over the Ethernet is handled much the same way. When the server stub returns to the Receiver, it calls the server's Sender procedure to transmit the result packet back to the caller machine. Once the result packet is queued for transmission, the server thread returns to the Receiver and again registers itself in the call table and waits for another call packet to arrive.

Back at the caller machine, the Ethernet interrupt routine validates the arriving result packet, does the UDP checksum, and tries to find the caller thread waiting in the call table. If successful, the interrupt routine directly awakens the caller thread, which returns to step 4 in the caller stub described above.

The steps involved in transporting a call packet and a result packet are nearly identical, from calling the Sender through transmitting and receiving the packet to awakening a suitable thread in the call table. We refer to these steps as the "send+receive" operation. A complete remote procedure call requires two send+receives -- one for the call packet and one for the result packet.

3.2 Structuring for Low Latency

The scenario just outlined for the fast path of RPC incorporates several design features that lower latency. We already mentioned that the stubs use custom generated assignment statements in most cases to marshall arguments and results for each procedure, rather than library procedures or an interpreter. Another performance enhancement in the caller stub is invoking the chosen Starter, Transporter, and Ender procedures through procedure variables filled in at binding time, rather than finding the procedures by a table lookup.

Directly awakening a suitable thread from the Ethernet interrupt routine is another important performance optimization for RPC. This approach means that demultiplexing of RPC packets is done in the interrupt routine. The more traditional approach is to have the interrupt handler awaken a datalink thread to demultiplex the incoming packet. The

---

[1] We lump 3 procedures of the actual implementation under the name Receiver here.

traditional approach lowers the amount of processing in the interrupt handler, but doubles the number of wakeups required for an RPC. As wakeups tend to be expensive, we prefer to avoid extra ones. By carefully coding the demultiplexing code for RPC packets, the time per packet in the interrupt handler can be kept within reasonable bounds (see Table VI). Even with only two wakeups for each RPC, the time to do these wakeups can be a major contributor to RPC latency. Considerable work has been done on the Firefly scheduler to minimize this cost. The slower traditional path through the datalink modules in the operating system address space is used when the interrupt routine cannot find the appropriate RPC thread in the call table, and when handling non-RPC packets.

The packet buffer management scheme we have adopted also increases RPC performance. We already mentioned above that the server stub hangs on to the call packet to use it for the results. We also arrange for the receive interrupt handler to immediately replace the buffer used by an arriving call or result packet. Each call table entry occupied by a waiting thread also contains a packet buffer. In the case of a calling thread it is the call packet; in the case of a server thread it is the last result packet. These packets must be retained for possible retransmission. The RPC packet exchange protocol is arranged so that arrival of a result or call packet means that the packet buffer in the matching call table entry is no longer needed. Thus, when putting the newly arrived packet into the call table, the interrupt handler removes the buffer found in that call table entry and adds it to the Ethernet controller's receive queue. Since the interrupt handler always checks for additional packets to process before terminating, on-the-fly receive buffer replacement can allow many packets to be processed per interrupt. Recycling is sufficiently fast that we have seen several hundred packets processed in a single receive interrupt.

The alert reader will have suspected another feature of our buffer management strategy: RPC packet buffers reside in memory shared among all user address spaces and the Nub. These buffers also are permanently mapped into Vax I/O space. Thus, RPC stubs in user spaces, and the Ethernet driver code and interrupt handler in the Nub, all can read and write packet buffers in memory using the same addresses. This strategy eliminates the need for extra address mapping operations or copying when doing RPC. While its insecurity makes shared buffers unsuitable for use in a time sharing system, security is acceptable for a single user workstation or for a server where only trusted code executes (say a file server). This technique would also work for, say, kernel to kernel RPCs. For user space to user space RPCs in a time sharing environment, the more secure buffer management required would introduce extra mapping or copying operations into RPCs.

Like the pool of RPC buffers, the RPC call table also is shared among all user address spaces and the Nub. The shared call table allows the Ethernet interrupt handler to find and awaken the waiting (calling or server) thread in any user address space.

Several of the structural features used to improve RPC performance collapse layers of abstraction in a somewhat unseemly way. Programming a fast RPC is not for the squeamish.

3.3 Allocation of Latency

We now try to account for the time an RPC takes. Table VI shows a breakdown of time for the send+receive operation that is executed twice per RPC, once for the argument packet, once for the result packet. The first seven actions are activities of the sending machine. The next three are Ethernet and hardware controller delays. The last four are actions performed by the receiving machine. All of the software in this table is written in assembly language.

Table VI: Latency of steps in the send+receive operation

Action	Microseconds for 74 byte packet		Microseconds for 1514 byte packet (if different)	
Finish UDP header (Sender)	59	**a**		
Calculate UDP checksum	45	**b**	440	**b**
Handle trap to Nub	37	**a**		
Queue packet for transmission	39	**a**		
Interprocessor interrupt to CPU 0	10	**c**		
Handle interprocessor interrupt	76	**a**		
Activate Ethernet controller	22	**a**		
QBus/Controller transmit latency	70	**d**	815	**e**
Transmission time on Ethernet	60	**d**	1230	**e**
QBus/Controller receive latency	80	**d**	835	**e**
General I/O interrupt handler	14	**a**		
Handle interrupt for received pkt	177	**a**		
Calculate UDP checksum	45	**b**	440	**b**
Wakeup RPC thread	220	**a**		
Total for send+receive	954		4414	

Key for Table VI:

- **a** Calculated by adding the measured execution times of the machine instructions in this code sequence.
- **b** Measured by disabling UDP checksums and noting speedup.
- **c** Estimated.
- **d** Measured with logic analyzer.
- **e** Measurements **d** adjusted assuming 10 megabit/second Ethernet, 16 megabit/second QBus transfer, and no cut through.

Table VI shows that Ethernet transmission time and QBus/controller latency are dominant for large packets, but software costs are dominant for small packets. The biggest single software cost is the scheduler operation to awaken the waiting RPC thread.

Table VII shows where time is spent in the user space RPC runtime code and standard stubs for a call to Null(). The procedures detailed in Table VII are written in Modula-2+. The times were calculated by adding up the instruction timings for the compiler-generated code.

Table VII: Latency of stubs and RPC runtime

Machine	Procedure	Microseconds
Caller	Calling program (loop to repeat call)	16
	Calling stub (call & return)	90
	Starter	128
	Transporter (send call pkt)	27
Server	Receiver (receive call pkt)	158
	Server stub (call & return)	68
	Null (the server procedure)	10
	Receiver (send result pkt)	27
Caller	Transporter (receive result pkt)	49
	Ender	33
	TOTAL	606

The Modula-2+ code listed in Table VII includes 9 procedure calls. Since each call/return takes about 15 microseconds, depending on the number of arguments, about 20% of the time here is spent in the calling sequence.

In Table VIII we combine the numbers presented so far to account for the time required for a complete call of Null() and of MaxResult(b).

Table VIII: Calculation of latency for RPC to Null() and MaxResult(b)

Procedure	Action	Microseconds
Null()	Caller, server, stubs and RPC runtime	606
	Send+receive 74-byte call packet	954
	Send+receive 74-byte result packet	954
	TOTAL	2514
MaxResult(b)	Caller, server, stubs and RPC runtime	606
	Marshall a 1440-byte VAR OUT result	550
	Send+receive 74-byte call packet	954
	Send+receive 1514-byte result packet	4414
	TOTAL	6524

The best measured total latency for a call to Null() is 2645 microseconds, so we've failed to account for 131 microseconds. The best measured total latency for a call to MaxResult(b) is 6347 microseconds, so we've accounted for 177 microseconds too much. By adding the time of each instruction executed and of each hardware latency encountered, we have accounted for the total measured time of RPCs to Null() and MaxResult(b) to within about 5% .

4. IMPROVEMENTS

One of the important steps in improving the performance of Firefly RPC over its initial implementation was to rewrite the Modula-2+ versions of the fast path code in the Ethernet send+receive operation in VAX assembly language. In this section we illustrate the speed-ups achieved by using machine code.

We then use the analysis and measurement reported so far to estimate the impact that other changes could have on overall RPC performance. It is hard to judge how noticeable these possible improvements would be to the normal user of the system. The Firefly RPC implementation has speeded up by a factor of three or so from its initial implementation. This improvement has produced no perceived change in the behavior of most applications, since the throughput is still limited by I/O devices. However, lower latency RPC may encourage programmers to use RPC interfaces where they might previously have been tempted to use *ad hoc* protocols before, and encourage the designers of new systems to make extensive use of RPC.

4.1 Assembly Language vs. Modula-2+

In order to give some idea of the improvement obtained when Modula-2+ code fragments are recoded in assembly language, the following table shows the time taken by one particular code fragment at various stages of optimization. This fragment was chosen because it was the largest one that was recoded and is typical of the improvements obtained for all the code that was rewritten.

Table IX: Execution time for main path of the Ethernet interrupt routine

Version	Time in microseconds
Original Modula-2+	758
Final Modula-2+	547
Assembly language	177

The "Original Modula-2+" was the state of the code before any assembly language code was written. The interrupt routine had already been carefully written for speed (we thought). The "Final Modula-2+" code was structured so that the compiler output would follow the assembly language version as closely as possible. Writing in assembly language is hard work and also makes the programs harder to maintain. Because RPC is the preferred communication paradigm for the Firefly, however, it seemed reasonable to concentrate considerable attention on its key code sequence. (There can't be too much assembly language in the fast path, or it wouldn't be fast!) The Modula-2+ compiler used here doesn't generate particularly good code. The speedup achieved by using assembly language might be less dramatic starting from a different compiler, but would still be substantial.

4.2 Speculations on Future Improvements

While improving the speed of the RPC system, we noticed several further possibilities for improving its performance and also considered the impact that faster hardware and networks would have. In this section we speculate on the performance changes such improvements might generate. Some of these changes have associated disadvantages or require unusual hardware. For each change, we give the estimated speedup for a call to Null() and a call to MaxResult(b). The effect on maximum throughput has not been estimated for all the changes, since this figure is likely to be limited by a single hardware component.

Some estimates are based on "best conceivable" figures, and these may ignore some practical issues. Also, the effects discussed are not always independent, so the performance improvement figures cannot always be added.

4.2.1. Different network controller

A controller which provided maximum conceivable overlap between Ethernet and QBUS transfers would save about 300 microseconds on Null() (11%), and about 1800 microseconds (28%) on MaxResult(b). It is more difficult to estimate the improvement in throughput with multiple threads, since the Ethernet controller is already providing some overlap in that case. We think improvement is still possible on the transmission side, since the saturated reception rate is 40% higher than the corresponding transmission rate (see section 5.1).

4.2.2. Faster network

If the network ran at 100 megabits/second and all other factors remained constant, the time to call Null() would be reduced by 110 microseconds (4%) and the time to call MaxResult(b) would reduce by 1160 microseconds (18%).

4.2.3. Faster CPUs

If all processors were to increase their speed by a factor of 3, the time to call Null() would reduce by about 1380 microseconds (52%). The time to call MaxResult(b) would reduce by 2280 microseconds (36%).

4.2.4. Omit UDP checksums

Omitting UDP checksums saves 180 microseconds (7%) on a call to Null() and 1000 microseconds (16%) on a call to MaxResult(b). At present, we use these end-to-end software checksums because the Ethernet controller occasionally makes errors after checking the Ethernet CRC. End-to-end checksums still would be essential for crossing gateways in an internet.

4.2.5. Redesign RPC protocol.

We estimate that by redesigning the RPC packet header to make it easy to interpret, and changing an internal hash function, it would be possible to save about 200 microseconds per RPC. This represents approximately 8% of a call to Null() and 3% of a call to MaxResult(b).

4.2.6. Omit layering on IP and UDP

We estimate that direct use of Ethernet datagrams, omitting the IP and UDP headers, would save about 100 microseconds per RPC, assuming that checksums were still calculated. This is about 4% of a call to Null() and 1-2% of a call to MaxResult(b). This change would make it considerably more difficult to implement RPC on machines where we do not have access to the kernel. It would also make it impossible to use RPC via an IP gateway. (Some of the fields in IP and UDP headers are actually useful, and would have to be incorporated into the RPC header.)

4.2.7. Busy wait

If caller and server threads were to loop in user space while waiting for incoming packets, the time for a wakeup via the Nub would be saved at each end. This is about 440 microseconds per RPC, which is 17% of a call to Null() and 7% of a call to MaxResult(b). Allowing threads to busy wait (in such a way that they would relinquish control whenever the scheduler demanded) would require changes to the scheduler and would make it difficult to measure accurately CPU usage for a thread.

4.2.8. Recode RPC runtime routines (except stubs)

If the RPC runtime routines in Table VII were rewritten in hand-generated machine code, we would expect to save approximately 280 microseconds per RPC. This corresponds to 10% of a call to Null() and 4% of a call to MaxResult(b). This figure is based on an expected speedup of a factor of 3 in 422 microseconds of routines to be recoded, which is typical of other code fragments that have been rewritten.

5. FEWER PROCESSORS

The Fireflies used in the tests reported here had 5 MicroVAX II processors. The measurements in other sections were done with all 5 available to the scheduler. In this section we report the performance when the number of available processors is decreased.

At first we were unable to get reasonable performance when running with a single available processor on the caller and server machines. Calls to Null() were taking around 20 milliseconds. We finally discovered the cause to be a few lines of code that slightly improved multiprocessor performance but had a dramatic negative effect on uniprocessor performance. The good multiprocessor code tends to lose about 1 packet/second when a single thread calls Null() using uniprocessors, producing a penalty of about 600 milliseconds waiting for a retransmission to occur. Fixing the problem requires swapping the order of a few statements at a penalty of about 100 microseconds for multiprocessor latency. The results reported in this section were measured with the

swapped lines installed. (This change was not present for results reported in other sections.)

These measurements were taken with the RPC Exerciser, which uses hand-produced stubs that run faster than the standard ones (because they don't do marshalling, for one thing). With the RPC Exerciser, the latency for Null() is 140 microseconds faster and the latency for MaxResult(b) is 600 microseconds faster than reported in Table I. Such hand-produced stubs might be used in performance-sensitive situations, such as kernel-to-kernel RPCs, where one could trust the caller and server code to reference all arguments and results directly in RPC packet buffers.

Table X: Calls to Null() with varying numbers of processors

caller processors	server processors	seconds for 1000 calls
5	5	2.69
4	5	2.73
3	5	2.85
2	5	2.98
1	5	3.96
1	4	3.98
1	3	4.13
1	2	4.21
1	1	4.81

Table X shows 1 thread making RPCs to Null(), with varying numbers of processors available on each machine. When the calls are being done one at a time from a single thread, reducing the number of caller processors from 5 down to 2 increases latency only about 10%. There is a sharp jump in latency for the uniprocessor caller. Reductions in server processors seem to follow a similar pattern. Latency with uniprocessor caller and server machines is 75% longer than for 5 processor machines.

Table XI: Throughput in megabits/second of MaxResult(b) with varying numbers of processors

caller processors server processors	5 5	1 5	1 1
1 caller thread	2.0	1.5	1.3
2 caller threads	3.4	2.3	2.0
3 caller threads	4.6	2.7	2.4
4 caller threads	4.7	2.7	2.5
5 caller threads	4.7	2.7	2.5

Table XI shows the effect on the data transfer rate of varying the number of processors on RPCs to MaxResult(b). In this test each thread made 1000 calls. Apparently throughput is quite sensitive to the difference between a uniprocessor and a multiprocessor. Uniprocessor throughput is slightly more than half of 5 processor performance for the same number of caller threads.

We haven't tried very hard to make Firefly RPC perform well on a uniprocessor machine. The fast path for RPC is followed exactly only on a multiprocessor. On a uniprocessor, extra code gets included in the basic latency for RPC, such as a longer path through the scheduler. It seems plausible that better uniprocessor throughput could be achieved by an RPC design, like Amoeba's [7], V's [3], or Sprite's [6], that streamed a large argument or result for a single call in multiple packets, rather than depended on

multiple threads transferring a packet's worth of data per call. The streaming strategy requires fewer thread-to-thread context switches.

6. OTHER SYSTEMS

A sure sign of the coming of age of RPC is that others are beginning to report RPC performance in papers on distributed systems. Indeed, an informal competition has developed to achieve low latency and high throughput. Table XII collects the published performance of several systems of interest. All of the measurements were for inter-machine RPCs to the equivalent of Null() over a 10 megabit Ethernet, with the exception that the Cedar measurements used a 3 megabit Ethernet. No other paper has attempted to account exactly for the measured performance, as we have tried to do.

Table XII: Performance of remote RPC in other systems

System	Machine - Processor	~ MIPs	Latency in milliseconds	Throughput in megabits/sec
Cedar [2]	Dorado - custom	1 x 4	1.1	2.0
Amoeba [7]	Tadpole - M68020	1 x 1.5	1.4	5.3
V [3]	Sun 3/75 - M68020	1 x 2	2.5	4.4
Sprite [6]	Sun 3/75 - M68020	1 x 2	2.8	5.6
Amoeba/Unix [7]	Sun 3/50 - M68020	1 x 1.5	7.0	1.8
Firefly	FF - MicroVAX II	1 x 1	4.8	2.5
Firefly	FF - MicroVAX II	5 x 1	2.7	4.6

Amoeba advertises itself as the world's fastest distributed system. But the Cedar system achieved 20% lower latency 4 years earlier (using a slower network and a faster processor). Determining a winner in the RPC sweepstakes is tricky business. These systems vary in processor speed, I/O bus bandwidth, and controller performance. Some of these RPC implementations work only kernel to kernel, others work user space to user space. Some protocols provide internet headers, others work only within a single Ethernet. Some use automatically generated stubs, others use hand-produced stubs. Some generate end-to-end checksums with software, others do not. The implementations are written in different languages with varying quality compilers. It is not clear which corrections to apply to normalize the reported performance of different systems.

It is clear that developers of distributed systems are learning how to get good request/response performance from their system and network. It is now widely understood that it is not necessary to put up with high latency or low throughput from RPC-style communication. Some RPC implementations appear to drive current Ethernet controllers at their throughput limit[1] and to provide basic remote call latency only about 100 times slower than that for statically- linked calls within a single address space.

7. CONCLUSIONS

Our objective in making Firefly RPC the primary communication mechanism between address spaces, both inter-machine and local, was to explore the bounds of effectiveness of this paradigm. In making the RPC implementation fast, we sought to remove one excuse for not using it. To make it fast we had to understand exactly where time was being spent, remove unnecessary steps from the critical path, give up some structural

[1] In the case of Firefly RPC, we noticed that throughput has remained the same as the last few performance improvements were put in place. The CPU utilization continued to drop as the code got faster.

elegance, and write key steps in hand-generated assembly code. We did not find it necessary to sacrifice function; RPC still allows multiple transports, works over wide area networks, copes with lost packets, handles a large variety of argument types including references to garbage collected storage, and contains the structural hooks for authenticated and secure calls. The performance of Firefly RPC is good enough that application and system designers accept it as the standard way to communicate.

The throughput of several RPC implementations (including ours) appears limited by the network controller hardware; a controller that provided more overlap between the I/O bus and the Ethernet would lower Firefly RPC latency 10% to 30%. The software overheads of RPC in the Firefly are within a factor of 2 of the level where no further effort should be expended to lower them for communication using a 10 megabit/second Ethernet. If, as expected, 100 megabit/second networks become a reality over the next few years then we may face the challenge of speeding up the software once again. Faster processors might not do the entire job.

8. ACKNOWLEDGEMENTS

Andrew Birrell designed the Firefly RPC facility and Sheng Yang Chiu designed the driver for the Ethernet controller. Andrew and Sheng Yang, along with Mike Burrows, Eric Cooper, Ed Lazowska, Sape Mullender, Mike Schroeder, and Ted Wobber have participated in the implementation and improvement of the facility. The recent round of performance improvements and measurements were done by Mike Burrows and Mike Schroeder, at Butler Lampson's insistence. The RPC latency includes two wakeup calls to the scheduler, whose design and implementation was done by Roy Levin. Andrew Birrell, Mark Brown, David Cheriton, Ed Lazowska, Hal Murray, John Ousterhout, and Susan Owicki made several suggestions for improving the presentation of the paper.

9. REFERENCES

1. Brian Bershad et al., "Lightweight Remote Procedure Call," Technical Report 89-04-02, Dept. of Computer Science, Univ. of Washington.
2. Andrew D. Birrell and Bruce Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, Feb. 1984 pp. 39-59.
3. David R. Cheriton, "The V Distributed System," Communications of the ACM, Mar 1988, pp 314-333.
4. Digital Equipment Corp., "DEQNA ETHERNET -- User's Guide," Sep 1986.
5. Digital Equipment Corp., "Microsystems Handbook ," 1985, Appendix A.
6 John K. Ousterhout et al., "The Sprite Network Operating System," Computer, Feb 1988, pp. 23-35.
7. Robbert van Renesse, Hans van Staveren, and Andrew S. Tannenbaum, "Performance of the World's Fastest Distributed Operating System," Operating Systems Review , Oct 1988, pp. 25-34.
8 Paul R. Rovner, "Extending Modula-2 to Build Large, Integrated Systems," IEEE Software, Nov 1986, pp 46-57.
9. C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite Jr., "Firefly: A Multiprocessor Workstation," IEEE Transactions on Computers, Aug 1988, pp 909-920.