The Peregrine
High-Performance RPC System

David B. Johnson[1]
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

(713) 527-4834

dbj@cs.cmu.edu, willy@cs.rice.edu

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-9116343, and by the Texas Advanced Technology Program under Grant No. 003604014.

[1]Author's current address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891.

Summary

The Peregrine RPC system provides performance very close to the optimum allowed by the hardware limits, while still supporting the complete RPC model. Implemented on an Ethernet network of Sun-3/60 workstations, a null RPC between two user-level threads executing on separate machines requires 573 microseconds. This time compares well with the fastest network RPC times reported in the literature, ranging from about 1100 to 2600 microseconds, and is only 309 microseconds above the measured hardware latency for transmitting the call and result packets in our environment. For large multi-packet RPC calls, the Peregrine user-level data transfer rate reaches 8.9 megabits per second, approaching the Ethernet's 10 megabit per second network transmission rate. Between two user-level threads on the same machine, a null RPC requires 149 microseconds. This paper identifies some of the key performance optimizations used in Peregrine, and quantitatively assesses their benefits.

Keywords: Peregrine, remote procedure call, interprocess communication, performance, distributed systems, operating systems

1 Introduction

The Peregrine remote procedure call (RPC) system is heavily optimized for providing high-performance interprocess communication, while still supporting the full generality and functionality of the RPC model [3, 10], including arguments and result values of arbitrary data types. The semantics of the RPC model provides ample opportunities for optimizing the performance of interprocess communication, some of which are not available in message-passing systems that do not use RPC. This paper describes how Peregrine exploits these and other opportunities for performance improvement, and presents Peregrine's implementation and measured performance. We concentrate primarily on optimizing the performance of *network* RPC, between two user-level threads executing on separate machines, but we also support efficient *local* RPC, between two user-level threads executing on the same machine. High-performance network RPC is important for shared servers and for parallel computations executing on networks of workstations.

Peregrine provides RPC performance that is very close to the *hardware* latency. For network RPCs, the hardware latency is the sum of the *network penalty* [6] for sending the call and the result message over the network. The network penalty is the time required for transmitting a message of a given size over the network from one machine to another, and is measured without operating system overhead or interrupt latency. The network penalty is greater than the *network transmission time* for packets of the same size because the network penalty includes additional network, device, and processor latencies involved in sending and receiving packets. Latency for local RPCs is determined by the processor and memory architecture, and includes the expense of the required local procedure call, kernel trap handling, and context switching overhead [2].

We have implemented Peregrine on a network of Sun-3/60 workstations, connected by a 10 megabit per second Ethernet. These workstations each use a 20-megahertz Motorola MC68020 processor and an AMD Am7990 LANCE Ethernet network controller. The implementation uses an RPC packet protocol similar to Cedar RPC [3], except that a blast protocol [20] is used for multi-packet messages. The RPC protocol is layered directly on top of the IP Internet datagram protocol [13]. In this implementation, the measured latency for a null RPC with no arguments or return values between two user-level threads executing on separate Sun-3/60 workstations on the

Ethernet is 573 microseconds. This time compares well with the fastest null network RPC times reported in the literature, ranging from about 1100 to 2600 microseconds [3, 12, 8, 15, 17, 19], and is only 309 microseconds above the measured *hardware* latency defined by the network penalty for the call and result packets in our environment. A null RPC with a single 1-kilobyte argument requires 1397 microseconds, showing an increase over the time for null RPC with no arguments of just the network transmission time for the additional bytes of the call packet. This time is 338 microseconds above the network penalty, and is equivalent to a user-level data transfer rate of 5.9 megabits per second. For large multi-packet RPC calls, the network user-level data transfer rate reaches 8.9 megabits per second, achieving 89 percent of the hardware network bandwidth and 95 percent of the maximum achievable transmission bandwidth based on the network penalty. Between two user-level threads executing on the same machine, a null RPC with no arguments or return values requires 149 microseconds.

In Section 2 of this paper, we present an overview of the Peregrine RPC system. Section 3 discusses some of the key performance optimizations used in Peregrine. In Section 4, we describe the Peregrine implementation, including single-packet network RPCs, multi-packet network RPCs, and local RPCs. The measured performance of Peregrine RPC is presented in Section 5. In Section 6, we quantify the effectiveness of the optimizations mentioned in Section 3. Section 7 compares our work to other RPC systems, and Section 8 presents our conclusions.

2 Overview of the Peregrine RPC System

The Peregrine RPC system follows the conventional RPC model of servers *exporting* one or more interfaces, making a set of procedures available, and clients *binding* to an interface before performing calls [3, 10]. Calls to these procedures appear to the client as a conventional procedure call and to the server as a conventional procedure invocation. In reality, the client invokes a client *stub* procedure that packages the call arguments and the identification of the procedure into a call message, which is then delivered to the server machine. Collecting the arguments into the message is referred to as *marshaling* [10]. Once at the server machine, a server *stub* procedure *unmarshals* the arguments from the call message, invokes the appropriate procedure in the server, and on return from that procedure, marshals the return values into the result message, which is delivered to the client machine. The client stub then unmarshals the return values from the result message and returns to the client program. In Peregrine, it is the client stub's responsibility to convert the call arguments into the server's representation, and to convert the results back on return. All network packets and all processing at the server use only the server's data representation, thereby offloading any data representation conversion overhead from a server to its clients.

RPC binding initializes data structures in the client and server kernels for use on each call and return, and returns a *binding number* to the client program. The binding number provides a form of capability, ensuring that no RPCs can be executed without first binding. In addition to the binding number, a word of *binding flags* is returned, identifying the data representation used by the server machine.

No special programming is necessary in the client or server, and no compiler assistance is required. In performing an RPC, argument values are pushed onto the call stack by the client as if calling the server routine directly. The binding number is stored by the client stub and is automatically provided to the kernel on each call. Although this associates a single server binding with the procedure names provided by the interface's client stubs, the associated binding number may also be changed before each call by the client if desired. The binding flags word is also stored by the client stub and is used by the stub on each call.

The client and server stubs are automatically generated from a description language similar to ANSI C function prototypes. Procedure arguments may be of any data type. Individual arguments may be either immediate data values or pointers to data values. Pointer arguments may optionally be declared in one of three ways, depending on whether the data value must be passed on the call, the return, or both [15]:

- An *in* pointer argument describes a data value that must be passed on the call, but need not be passed back on the return.

- An *out* pointer argument describes a data value that must be passed on the return, but need not be passed on the call.

- An *in-out* pointer argument describes a data value that must be passed on both call and return.

If not specified, a pointer argument is assumed to be *in-out*.

3 Optimizing RPC Performance

This paper concentrates on the following key optimizations used in the Peregrine RPC implementation:

1. Arguments (results) are transmitted directly from the user address space of the client (server), avoiding any intermediate copies.

2. No data representation conversion is done for argument and result types when the client and the server use the same data representation.

3. Both call and return packets are transmitted using preallocated and precomputed header templates, avoiding recomputation on each call.

4. No thread-specific state is saved between calls in the server. In particular, the server thread's stack is not saved, and there is no register saving when a call returns or register restoring when a new call is started.

5. The arguments are mapped into the server's address space, rather than being copied.

6. Multi-packet arguments are transmitted in such a way that no copying occurs in the critical path. Copying is either done in parallel with network transmission or is replaced by page remapping.

The first three optimizations can be used in any message-passing system, while the last three depend on RPC semantics and could not be used in a (non-RPC) message passing system. The semantics of the RPC model requires only that the specified procedure be executed in the server's address space [3, 10]. There is no requirement that the threads that execute these procedure calls continue to exist after the call has returned. Although the server's state must be retained between separate calls to the same server, no *thread-specific* state such as register or stack contents of these threads need be retained. Furthermore, the arguments for a new incoming call can be located at any available address in the server's address space.

4 Implementation

In Peregrine, the kernel is responsible for getting RPC messages from one address space to another, including fragmentation and reassembly, retransmission, and duplicate detection for messages that must travel across the network. The kernel also starts a new thread in the server when a call message arrives, and unblocks the client thread when the return message arrives. All processing specific to the particular server procedure being called is performed in the stubs, simplifying the kernel design.

The current Peregrine implementation uses the thread and memory management facilities of the V-System [6], but is not specific to the V-System and does *not* use the V-System's message passing primitives or protocols, with one exception. At bind time, Peregrine currently uses the V-System's facilities for connecting to a server. Additional traps were added to the kernel to support RPC call and return, and minor changes were made in several kernel routines, such as context switching and the Ethernet device driver. Most of the code added to the kernel has been written in C. Less than 100 assembly language instructions were added, consisting mainly of first-level trap and interrupt handling (before calling a C procedure) and context switching support.

4.1 Hardware Requirements

The Peregrine implementation utilizes the "gather" DMA capability of the Ethernet controller. Given a list of segments, each specified by a start address and a length, the Ethernet controller can transmit a single packet consisting of the combined data from those segments. The gather capability avoids copying noncontiguous segments of a call or result packet into a contiguous buffer before transmission. The ability to perform gather DMA is a common feature of many modern network interfaces and is not unique to the Sun architecture or the LANCE Ethernet controller.

We also rely on the ability to remap memory pages between address spaces by manipulating the page table entries. We place no restrictions on the relationship between the relative sizes of memory pages and network packets. In the Sun-3/60 architecture, remapping pages requires only the modification of the corresponding page table entries. However, many other architectures also require that the translation-lookaside buffer (TLB) entries for the remapped pages in the MMU be modified. Page remapping can still be performed efficiently in such systems with modern MMU designs. For example, the new Sun Microsystems SPARC reference MMU [14, 18] uses a TLB but allows individual TLB entries to be flushed by virtual address, saving the expense of reloading the entire TLB after a global flush. Similarly, the MIPS MMU [9] allows the operating system to individually modify any specified TLB entry.

4.2 The Packet Header

The Peregrine RPC protocol is layered directly on top of the Internet IP protocol [13], which in our current implementation is layered directly on top of Ethernet packets. Figure 1 shows the IP and Peregrine RPC packet headers. In the RPC header, the packet type is either "call" or "return." The

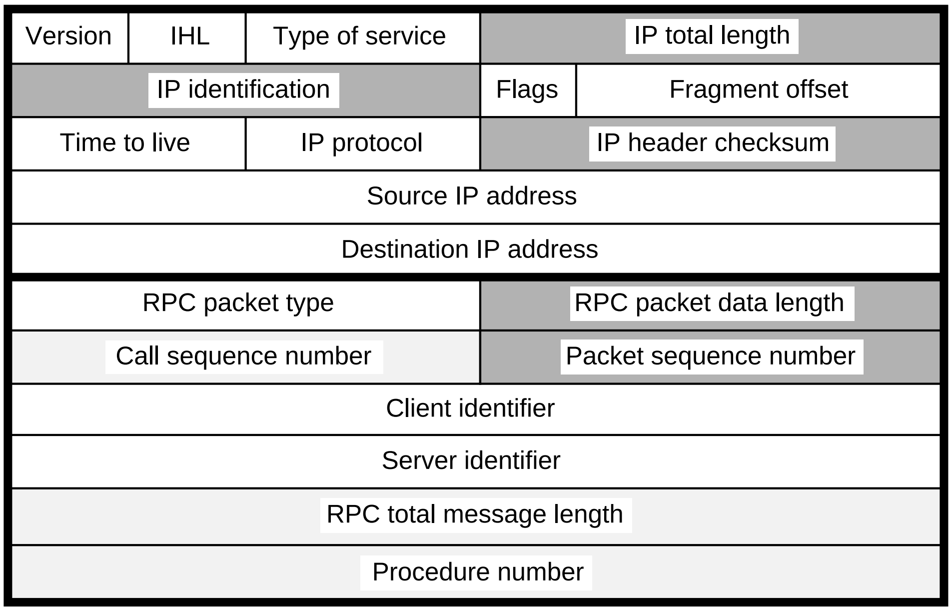

Figure 1 IP and Peregrine RPC packet headers

call sequence number identifies separate RPCs from the same client, whereas the packet sequence number identifies each packet of a particular call or result message. For single-packet argument or result messages, the packet sequence number is always 0. The procedure number identifies the particular procedure to call in the server's address space, and is used only for call packets. The length of the data present in this packet is specified in the RPC packet data length field, and the total length of all data being sent with this RPC call or return is specified in the RPC total message length field.

The headers of the call packets sent from one client to a particular server change little from one packet to the next. Likewise, the headers of the result packets sent by that server back to the same client change little between packets. Many of the fields of the Ethernet header, the IP header, and the RPC header can be determined in advance and reused on each packet transmission. At bind time, a packet header template is allocated and the constant fields are initialized; only the remaining fields are modified in sending each packet. In particular, the entire Ethernet header remains constant for the duration of the binding. In the IP and RPC headers, only the heavily shaded fields indicated in Figure 1 are changed between packets. The lightly shaded fields change between calls, but not between individual packets of the same call. Furthermore, since most of the IP header remains the same between calls, most of the header can be checksummed at bind time, requiring only the few modified fields to be added to the checksum on each packet transmission.

4.3 Client and Server Stubs

The client stub consists mainly of a kernel trap to transmit the call message, followed by a subroutine return instruction. The server stub consists mainly of a subroutine call to the specified procedure, followed by a kernel trap to transmit the result message. This trap in the server stub does not return. Since a new server thread is created to handle each call, this trap instead terminates the thread and prepares it for reuse on the next call.

For each area of memory that must be transmitted with a call or result message, the stub builds a *buffer descriptor* containing the address and the length of that area. The client stub builds a descriptor for the stack argument list and for the area of memory pointed to by each pointer argument. The length of this area must be available to the stub. Usually, this length is also an argument to the procedure, but it may need to be computed by the stub. Each buffer descriptor is tagged to indicate to the kernel whether that buffer is *in*, *out*, or *in-out*. The client stub builds descriptors for all pointer arguments, but the server stub builds a descriptor only for the *out* and *in-out* pointer arguments. These descriptors built by the server stub are not tagged since they are all treated as *out* by the kernel in sending the result message.

At binding time, the binding flags word is set to indicate any data representation conversion necessary for communication between this particular client and server. If both client and server use the same data representation, no flags are set and all data representation conversion is bypassed. If representation conversion is necessary, the client stub creates new buffers on its stack to hold the data values in the server's representation. For example, the client stub may need to byte-reverse integers or convert floating point data formats for arguments being passed on the call. These stack buffers are deallocated automatically with the rest of the stack frame when the client stub returns, simplifying the client stub and avoiding the overhead of allocating these buffers on the heap. For the stack argument list itself, as a special case, any data representation conversion is performed "in place" if the client and server representations are the same size, replacing the old values with the same values in the server's representation. This special case is possible since, by the compiler's procedure calling convention, the stack arguments are scratch variables to the client stub and are automatically deallocated from the stack upon return.

4.4 Single-Packet Network RPC

4.4.1 Sending the Call Packet

When a process performs an RPC, the arguments are pushed onto the process's stack and a procedure call to the client stub is performed as if calling the server procedure directly. The client stub builds the buffer descriptors on the stack as described in Section 4.3, pushes the procedure number and the binding number onto the stack, and traps into the kernel. The stack contents at the time of the trap are illustrated in Figure 2. The kernel directs the Ethernet interface to transmit the packet using gather DMA from the packet header template corresponding to the

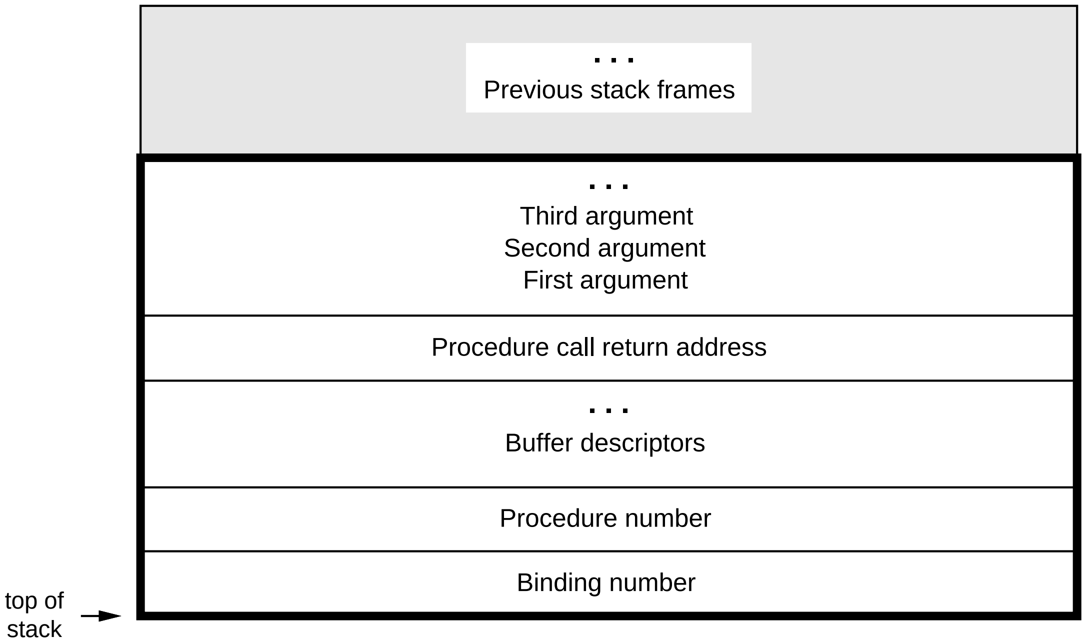

Figure 2 Client stack on entry to the kernel

given binding number and from the buffers indicated by the *in* and *in-out* descriptors. On the Sun architecture, however, I/O using DMA can proceed only to or from virtual addresses in kernel memory. Therefore, for each buffer other than the packet header template (which already resides in kernel memory), the corresponding memory pages are first double-mapped into kernel space before instructing the Ethernet interface to transmit the packet. The arguments are copy-on-write protected so that they are available for possible retransmission.

4.4.2 Receiving the Call Packet

The arguments for the call are received as part of the packet into a buffer in the kernel, but before calling the requested procedure in the server, the arguments must be placed on the top of the server thread's stack. To avoid copying them onto the stack, we arrange instead to use the packet buffer itself as the server thread's stack. At initialization time, the Ethernet interface is configured with a number of packet buffers, each with size equal to the maximum Ethernet packet size, such that each buffer is at the high-address end of a separate virtual memory page. The layout of the call packet in this page after receipt is shown in Figure 3.

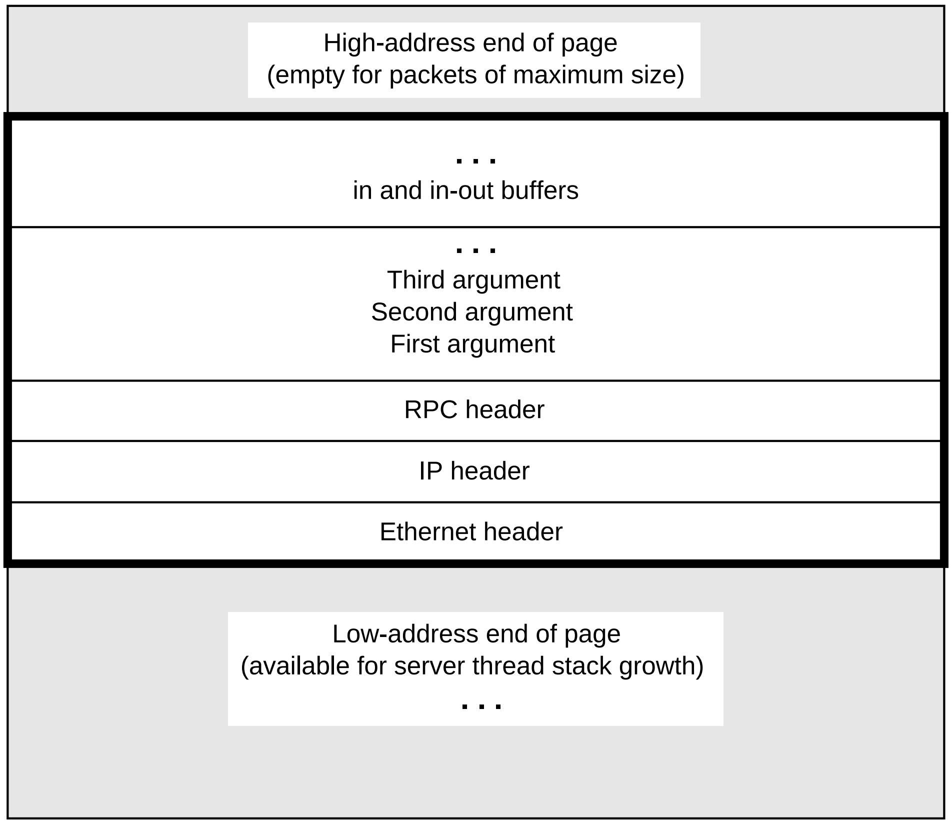

Figure 3 Received call packet in one of the server's Ethernet receive buffer pages

On arrival of a packet, the Ethernet interrupt routine examines the packet. The IP header checksum is verified, and if the packet is an RPC call packet, control is passed to the RPC packet handler. Duplicate detection is done using the RPC call and packet sequence numbers. A preallocated server thread is then reinitialized to execute this call, or if no free threads are available in the pool, a new thread is created. The receive packet buffer page is remapped into the server's address space at the location for the thread's stack, and an unused page is remapped to the original virtual address of the packet buffer to replace it for future packets. The thread begins execution at the server's RPC dispatch procedure that was registered when the server exported the RPC interface.

The arguments and procedure number from the remapped packet buffer are on the top of the thread's stack. The server dispatch procedure uses the procedure number to index into a table of procedure-specific stub addresses. It then pops all but the call arguments off its stack and jumps to the corresponding stub address from the table. If there are no pointer arguments, the corresponding user procedure in the server is immediately called by the stub. If there are pointer arguments, each must first be replaced with a pointer to the corresponding buffer in the server's address space. For *in* and *in-out* pointers, the address of the buffer that arrived with the packet, which is now on the server stack, is used. For *out* pointers, no buffer was sent in the packet, and a new buffer is created instead. The stub then calls the corresponding user procedure in the server.

4.4.3 Sending the Result Packet

When the server procedure returns to the stub, the stub builds any necessary buffer descriptors for *out* or *in-out* pointer arguments, as described in Section 4.3, and then traps into the kernel with the functional return value of the procedure in a register following the MC680x0 compiler convention. The result packet is then transmitted using the Ethernet interface gather DMA capability, in the same way as for the call packet (Section 4.4.1). Any result buffers are copy-on-write protected in the server's address space, to be available for possible retransmission. When the results are received by the client, an acknowledgement is returned and the result buffers are unprotected. As with the Cedar RPC protocol [3], the transmission of a new RPC by the same client to this server shortly after this reply also serves as an implicit acknowledgement. The server thread does not return from this kernel trap, and thus when entering the kernel on this trap, and later when leaving the kernel to begin a new RPC, the thread's registers need not be saved or restored.

4.4.4 Receiving the Result Packet

When the result packet is received by the client kernel, the copy-on-write protection is removed from the arguments. The functional return value is copied into the client thread's saved registers. Any data in the packet being returned for *out* or *in-out* buffers are copied by the kernel to the appropriate addresses in the client's address space, as indicated by the buffer descriptors built by the client stub before the call. The client thread is then unblocked and resumes execution in the client stub immediately after the kernel trap from the call.

The copy of the return buffers into the client's address space could be avoided by instead remapping the packet buffer there, but this would require a modification to conventional RPC semantics. Pointers to *out* or *in-out* buffers would need to be passed by the user instead as a *pointer to a pointer* to the buffer rather than directly as a single pointer, and the client stub would then overwrite the second pointer (pointing to the buffer) with the address of the buffer in the remapped packet, effectively returning the buffer without copying its data. The client would then be responsible for later deallocating memory used by the return packet buffer. This semantics is used by the Sun RPC implementation [17] to avoid the expense of copying. We have not implemented this mechanism in Peregrine because we want to preserve conventional RPC semantics.

4.5 Multi-Packet Network RPC

For a network RPC in which the message containing the argument or result values is larger than the data portion of a single Ethernet RPC packet, the message is broken into multiple packets for transmission over the network. These packets are sent using a *blast* protocol [20] to reduce latency by eliminating per-packet acknowledgements. Selective retransmission is used to avoid retransmission of the entire message. As in the single-packet case, the data are transmitted directly from the client's address space using gather DMA to avoid copying. Once the arguments (results) have been transmitted and assembled at the server (client) machine, the execution of a multi-packet network RPC is the same as for the single-packet case described in Section 4.4.

The call message is composed with the argument list and buffers in the same order as for a single-packet call, which is then transmitted in a sequence of packets each of maximum size. However, the packet that would conventionally be sent *first* (containing, at least, the beginning of the argument list) is instead sent *last*. That is, the first data packet transmitted starts at an offset into the call message equal to the maximum data size of a packet. Following this, the remainder of the packets are sent in their conventional order, followed finally by what would otherwise have been the first packet. For example, Figure 4 illustrates the transmission of a call message composed of four packets. The packets are transmitted in the order 1, 2, 3, 0: packet 0 contains the beginning of the argument list, and packet 3 contains the end of the last *in* or *in-out* buffer.

On receipt at the server machine, the packets are reassembled into the call message in a single contiguous buffer. For all but the last packet received (the packet containing the beginning of the

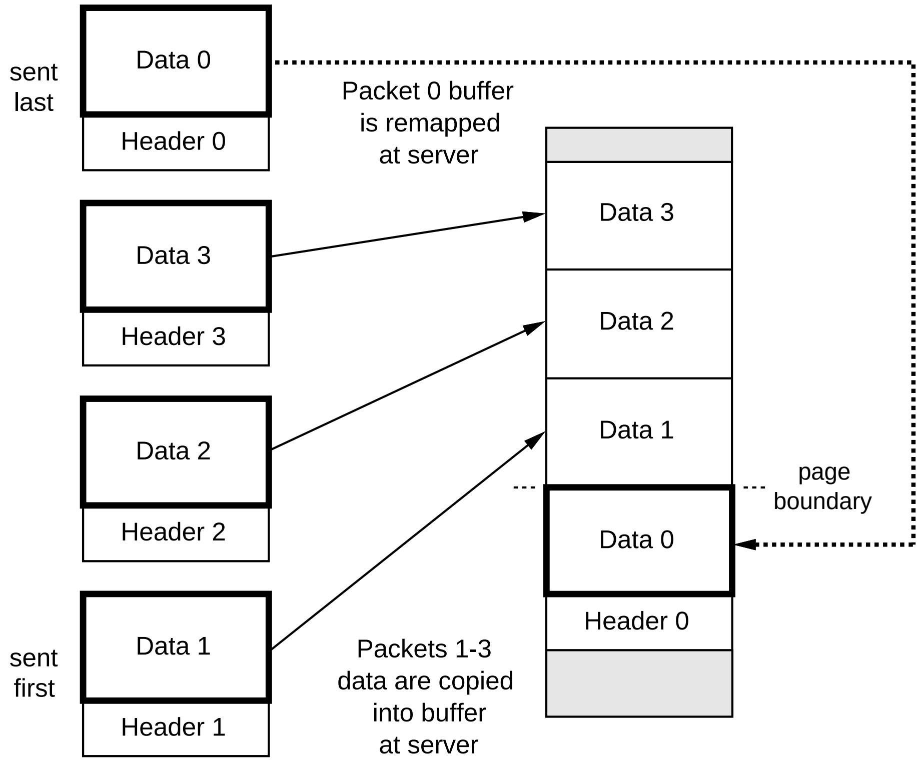

Figure 4 Example multi-packet call transmission and reception

argument list), the packet data are copied into the contiguous buffer, as illustrated in Figure 4. This buffer begins with the space for the data of packet 1 and is located on a page-aligned address. For each packet copied, the copying overhead occurs in parallel with the transmission of the following packet of the call. In Figure 4, the copying of the data from packet 1 at the server occurs in parallel with the transmission of packet 2 by the client, and so forth. When the last packet arrives (packet 0), it is instead remapped into the page located immediately before the assembled argument buffer (now containing the data of packets 1 through 3). As described in Section 4.4.2, the Ethernet receive buffers are each located at the high-address end of a separate virtual memory page. Thus, by remapping this last packet to a page immediately before the buffer containing the copied data from the other packets of the call, a contiguous argument buffer is assembled *without copying* the data of this final packet. Since this is the last packet transmitted for this call, the copying of this packet's data could not be done in parallel with the transmission of another packet of the call. By remapping this packet instead of copying its data, we avoid this overhead. Since there is unused space below the Ethernet receive buffer in each page, only the first packet of the original call message can be remapped in this way to assemble a contiguous buffer containing the arguments, therefore necessitating the transmission of the "first" packet last.

If the result message requires multiple packets, it is likewise sent as a sequence of packets using a blast protocol. As described in Section 4.4.4, however, the result data arriving at the client are always copied into the buffers described by the *out* and *in-out* descriptors built by the client stub, and no remapping is used. Thus, the packets of the result message are simply sent in their conventional order rather than the order used in transmitting a multi-packet call message. Once the complete result message has arrived, the client thread is restarted as in the single-packet case.

4.6 Local RPC

Between two threads executing on the same machine, Peregrine uses memory mapping to efficiently move the call arguments and results between the client's and server's address spaces. The technique used is similar to our remapping of the Ethernet packet receive buffer to form the server thread's stack for network RPCs. The execution of the call in the client and server stubs is the same as for network RPC. The client stub builds the buffer descriptors and traps into the kernel. Once in the kernel, if the server is local, the arguments are copied from the memory areas indicated by the *in* and *in-out* buffer descriptors into a page-aligned buffer in the client's address space, and this buffer is then remapped from the client's address space into the server's address space to become the stack for the server thread's execution of the call. A new thread from the pool of preallocated server threads created at bind time is reinitialized for this call, and begins execution in the server dispatch procedure. On return from the call, the stack buffer is remapped back into the client thread's address space, and the kernel then copies any *out* or *in-out* arguments to their correct locations in the client's address space. By remapping the buffer between address spaces, the arguments are never accessible in both the client and server address spaces at the same time. This prevents other threads in the client's address space from potentially modifying the arguments in the server during the call. Although this implementation of local RPC is similar to that used in the LRPC system [2], Peregrine differs significantly in several areas, as will be discussed in Section 7.

Copying the arguments before remapping them into the server's address space is necessary in order to preserve RPC semantics for arguments that do not fill an entire page. Without this copy, we would need to remap all pages of the client's address space that contained any portion of the argument list or data values described by pointer arguments. Since there may also be other data values in these same pages, this remapping would allow the server access to these values, allowing them to be read or modified and perhaps violating the correctness or security requirements of the client. Also, since the remapped pages are not accessible in the client's address space during the execution of the call by the server, other threads in the client's address space would not have access to these pages and could be forced to wait until the call completed before continuing their own execution. The copying is performed in the kernel rather than in the client stub in order to use the same stubs for both local and network RPC.

5 Performance

All performance measurements presented in this paper were obtained using diskless Sun-3/60 workstations connected by a 10 megabit per second Ethernet network. The network was otherwise idle during the measurements. For each individual measurement, the total elapsed time for a trial of 10,000 iterations (or 100,000 iterations for some measurements) was measured and divided by the number of iterations. Among individual trials for the same measurement, the results varied by less than 1 percent. For each measurement, the performance figures presented have been averaged over several individual trials.

5.1 The Network Penalty

The *network penalty* [6] is the minimum time required to transfer a given number of bytes over the network from one machine to another on an idle network. It is a function of the processor, the network, the network interface, and the number of bytes transferred. The network penalty does not include protocol overhead, context switching, or interrupt processing costs. It represents the cost of transferring the data in one direction only. Any acknowledgements required by particular higher-level protocols would incur a separate network penalty. Table 1 shows the measured network penalty for single-packet transfers, for various packet sizes ranging from the minimum to the maximum Ethernet packet size. Also shown in Table 1 are the network transmission times for the same packet sizes, computed at the Ethernet transmission rate of 10 megabits per second, and the difference between the transmission time and the network penalty. This difference is due to a number of factors, including additional network and device latencies and processor cost. The network latency stems from the transmission of synch and preamble bits, and from the delay in listening for the carrier. The device latency results from checking the LANCE Ethernet interface's *buffer descriptor rings*, used for communication between the CPU and the interface, at both the sender and the receiver. Additional latency occurs as a result of filling the DMA FIFO on the sender, and flushing the end of the packet to memory on the receiver. The interface furthermore checks the CRC of each incoming packet, and sets a number of bits in the device register. Similarly, on transmission the CPU sets a number of bits in the device register to direct the interface to send a packet.

For data transfers larger than the maximum Ethernet packet data size, the network penalty measures the cost of streaming the required number of packets over the network. The packets are sent as quickly as possible, with no delay between packets and no protocol acknowledgements.

Data Size (bytes)	Transmission Time * (microseconds)	Network Penalty (microseconds)	Overhead (microseconds)
46	51.2	132	80.8
100	94.4	173	78.6
200	174.4	255	80.6
400	334.4	417	82.6
800	654.4	743	88.6
1024	833.6	927	93.4
1500	1214.4	1314	99.6

*The transmission time includes 18 additional bytes in each packet consisting of the Ethernet destination address, source address, packet type, and CRC.

Table 1 Network penalty for various packet sizes

Table 2 shows the measured network penalty for multi-packet data transfers, for various multiples of the maximum Ethernet packet data size. The transmission data rate achieved is also shown. In addition to the reasons mentioned previously for the difference between network penalty and network transmission time for single-packet transfers, the interpacket gap required by the LANCE Ethernet controller for sending back-to-back transmissions prevents full utilization of the 10 megabit per second bandwidth of the Ethernet [4].

Data Size (bytes)	Number of Packets	Network Penalty (milliseconds)	Data Rate (Mbits/second)
1500	1	1.31	9.16
3000	2	2.58	9.30
6000	4	5.13	9.36
12000	8	10.21	9.40
24000	16	20.40	9.41
48000	32	40.83	9.40
96000	64	81.59	9.41

Table 2 Network penalty for multi-packet data transfers

5.2 Single-Packet Network RPC

The performance of the Peregrine RPC system for single-packet RPCs is close to the network penalty times given in Table 1. Table 3 summarizes our measured RPC performance. The network penalty shown represents the cost of sending the call and result packets over the network. The difference for each case between the measured RPC time and the corresponding network penalty times indicates the overhead added by Peregrine.

To determine the sources of this overhead, we also separately measured the execution times for various components of a null RPC. In this cost breakdown, each component was executed 100,000 times in a loop, and the results averaged. In a few cases, such as in measuring the execution time of the Ethernet interrupt routine, small changes were made to the timed version of the code in order to be able to execute it in a loop, but all such changes closely preserved the individual instruction execution times. These results are shown in Table 4. The components are divided between those that occur in the client before transmitting the call packet, those in the server between receiving the call packet and transmitting the result packet, and those in the client after receiving the result packet. A number of operations necessary as part of a network RPC do not appear in Table 4 because they occur in parallel with other operations. On the client's machine, these operations include putting the call packet on the retransmission queue, handling the Ethernet transmit interrupt for the call packet, blocking the calling thread, and the context switch to the next thread to run while waiting for the RPC results. On the server's machine, the operations that occur in parallel with other components of the RPC cost breakdown include handling the Ethernet transmit interrupt for the result packet, and the context switch to the next thread.

Procedure	Network Penalty	Measured	RPC Overhead
Null RPC	264	573	309
4-byte `int` argument RPC	267	583	316
1024-byte *in* RPC	1059	1397	338
1024-byte *in-out* RPC	1854	2331	477

Table 3 Peregrine RPC performance for single-packet network RPCs (microseconds)

Component	Time
Procedure call to client stub and matching return	2
Client stub	5
Client kernel trap and context switch on return	37
Client Ethernet and RPC header completion	2
Client IP header completion and checksum	5
Client sending call packet	41
Network penalty for call packet	132
Server Ethernet receive interrupt handling	43
Server Ethernet and RPC header verification	2
Server IP header verification and checksum	7
Duplicate packet detection	5
Page remapping of receive buffer to be server stack	4
Other server handling of call packet	8
Server context switch and kernel trap on return	26
Server stub	4
Server Ethernet and RPC header completion	2
Server IP header completion and checksum	5
Server sending result packet	25
Network penalty for result packet	132
Client Ethernet receive interrupt handling	43
Client Ethernet and RPC header verification	2
Client IP header verification and checksum	7
Client handling of result packet	5
Total measured cost breakdown	544
Measured cost of complete null RPC	573

Table 4 Measured breakdown of costs for a null network RPC (microseconds)

The cost of sending the call packet is more expensive than the cost of sending the result packet because the binding number must be checked and the corresponding binding data structures found within the kernel. Directing the Ethernet interface to transmit the call packet is also more expensive than for the result packet due to the provision for sending the packets of a multi-packet call message in a different order, as described in Section 4.5. Although this feature is not used on a null RPC, its presence does affect the null call's overhead.

The use of the IP Internet Protocol [13] in the Peregrine implementation adds a total of only 24 microseconds to the overhead on each null RPC. Large calls experience a further overhead of 16 microseconds in transmission time on the Ethernet for each IP header (at 10 megabits per second), but this overhead does not affect the null RPC time since its packet size is still less than the minimum allowed Ethernet packet size. Although the use of IP is not strictly necessary on a single local area network, it allows Peregrine packets to be forwarded through IP gateways, justifying its modest cost.

5.3 Multi-Packet Network RPC

Table 5 shows the performance of Peregrine RPC for various RPCs with large, multi-packet call or result messages. The throughput indicates the speed at which the argument and result values are transmitted, and does not include the size of the Ethernet, IP, or RPC packet headers. Like the single-packet network RPC performance, the performance of multi-packet network RPC is close to the network penalty for sending the call and return messages. Peregrine achieves a throughput of up to 8.9 megabits per second, coming within 89 percent of the network bandwidth and within 95 percent of the maximum bandwidth as limited by the network penalty.

5.4 Local RPC

Unlike network RPC performance, the performance of local RPC is determined primarily by processor overheads [2]. In particular, the minimum cost of a local null RPC is the sum of the costs of a procedure call from the client user program to the client stub, a kernel trap and a context switch on the call, and a kernel trap and a context switch on the return. In order to prevent the server from accessing other memory in the client and to prevent other threads in the client from

Procedure (bytes)	Network Penalty (milliseconds)	Latency (milliseconds)	Throughput (Mbits/second)
3000-byte *in* RPC	2.71	3.20	7.50
3000-byte *in-out* RPC	5.16	6.04	7.95
48000-byte *in* RPC	40.96	43.33	8.86
48000-byte *in-out* RPC	81.66	86.29	8.90

Table 5 Peregrine RPC performance for multi-packet network RPCs

modifying the arguments during the server's execution of the call, at least one memory-to-memory copy of the arguments is also required. The kernel trap and address space switch costs depend heavily on the processor architecture and operating system design [1]. The figures reported for these operations in Table 4 apply to the local RPC case as well, making the minimum null local RPC cost in our environment 65 microseconds.

Table 6 summarizes the measured performance of local RPC in the Peregrine implementation. Relative to the minimum null RPC cost, this measured performance shows an implementation overhead of 84 microseconds for a local null call. This overhead includes the costs of executing the client and server stubs, validating the binding number in the kernel, reinitializing a server thread to execute the call, remapping the argument buffer from the client address space into the stack for the new server thread, and remapping the buffer back to the client on return.

6 Effectiveness of the Optimizations

In this section, we revisit the six optimizations listed in Section 3 and discuss their effectiveness in the Peregrine RPC implementation:

1. Arguments (results) are transmitted directly from the user address space of the client (server), avoiding any intermediate copies.

   To determine the effectiveness of this optimization, we measured the cost of performing memory-to-memory copies using the standard copying routine available in our C runtime library. For small copies, the library routine is inefficient, requiring 12 microseconds to copy only 4 bytes (and 9 microseconds to copy 0 bytes), due to a high startup overhead for

Procedure	Time
Null RPC	149
4-byte `int` argument RPC	150
1024-byte *in* RPC	310
1024-byte *in-out* RPC	468

Table 6 Performance of local Peregrine RPC (microseconds)

determining different special cases for loop unrolling during the copy. To copy 1024 bytes, the library routine requires 159 microseconds, and to copy the user-level data of a maximum-sized Ethernet packet requires 209 microseconds. In contrast, using gather DMA to transmit the packet without memory-to-memory copies avoids these costs entirely, but adds 3 microseconds per page for the copy-on-write protection, plus between 3 and 7 microseconds per packet for setting up the Ethernet interface to transmit each separate area of memory to be sent as part of the packet. We conclude that even for the smallest argument and return value sizes, transmission from the user address space results in a performance gain.

2. No data representation conversion is done for argument and result types when the client and the server use the same data representation.

   RPC systems, such as SunRPC [17], that use a single "external representation" for data types must convert all arguments and result values to this standard representation, even if the client and server machines both use the same native representation that happens to differ from the standard RPC external representation. The savings achieved by not performing this conversion when the client and server machines use the same native representation depend on the size and type of argument and result values. As an example of these savings in our hardware environment, the time required to byte-swap a single 4-byte integer using the routine available in our C runtime library is 13 microseconds, and the time required to byte-swap an array of 256 integers (1024 bytes) is 1076 microseconds.

3. Both call and return packets are transmitted using preallocated and precomputed header templates, avoiding recomputation on each call.

   As shown in Table 4, the cost of transmitting the call packet is 48 microseconds, including the time to complete the IP and RPC headers. The corresponding cost of transmitting the result packet is 32 microseconds. To evaluate the effectiveness of using the header templates, we measured these same components in a modified version of the implementation that did not use this optimization. Based on these measurements, the packet header templates save 25 microseconds per packet, or 50 microseconds total for a null RPC. Of these 25 microseconds per packet, 7 microseconds are spent building the Ethernet and RPC headers, and 18 microseconds for the IP header and IP checksum.

4. No thread-specific state is saved between calls in the server. In particular, the thread's stack is not saved, and there is no register saving when a call returns or register restoring when a new call is is started.

   In our implementation, this optimization saves 11 microseconds per RPC, as shown by the differing kernel trap and context switch overhead of the client and the server threads in Table 4. The savings from this optimization occur entirely at the server, since the context switch for the client thread while waiting for the RPC results to be returned must be a complete context switch, saving and restoring all registers. On processors with larger numbers of registers that must be saved and restored on a context switch and a kernel trap, such as the SPARC processor's register windows [18], this optimization will increase further in significance [1].

5. The arguments are mapped into the server's address space, rather than being copied.

   The cost of performing memory-to-memory copies was reported above. From Table 4, the cost of remapping the Ethernet receive buffer in the server to become the new server thread's stack is 4 microseconds. Thus, even for a null RPC, remapping the stack saves 5 microseconds over the cost of calling the library's memory copying routine. Similarly, for a 4-byte `int` argument RPC, 8 microseconds is saved by remapping rather than copying the arguments. Since the cost of remapping the buffer is independent of the size of the argument list (up to the maximum packet size), the savings by remapping rather than copying increase quickly with the argument list size. For example, the performance gain achieved by remapping the arguments into the server's address space for a 1024-byte *in* RPC is 155 microseconds, and for a maximum-sized single packet RPC, 205 microseconds.

6. Multi-packet arguments are transmitted in such a way that no copying occurs in the critical path. Copying is either done in parallel with network transmission or is replaced by page remapping.

   For all but the last packet transmitted, the copying occurs in parallel with the transmission of the following packet on the network. Copying the data of a full packet requires 209 microseconds, whereas the network transmission time for a full packet is 1214 microseconds. Thus, this copying requires only about 16 percent of the minimum time between packets. The overhead of copying the data of the last packet of a multi-packet call message is also eliminated by remapping the packet into the server's address space, saving an additional 209 microseconds.

7 Comparison to Other Systems

Comparison of RPC systems is difficult because differences in hardware and software platforms present implementors with different design tradeoffs. All RPC systems, however, face a similar set of challenges in achieving good performance. First among these challenges is avoiding expensive copies both in the client and the server. Expensive data representation conversions and recomputation of headers must also be avoided to the extent possible. Reducing overhead for thread management in the server is another important concern. We summarize here the approaches used by a number of systems for which good performance has been reported in the literature [2, 5, 12, 8, 15, 19].

Some of the optimizations incorporated in Peregrine are similar to optimizations used by the LRPC system [2]. However, LRPC supports only *local* RPC, between two threads executing on the same machine, whereas we have concentrated primarily on *network* RPC support, between two threads executing on separate machines over the network. LRPC preallocates and initializes control and data structures at bind time to reduce later per-call latency. This preallocation is roughly similar to our preallocation of packet header templates at bind time. Also, like Peregrine, LRPC does not retain server thread-specific state between calls, although LRPC takes advantage of this in a different way than does Peregrine. LRPC uses the client thread directly in the server's address space to execute the call, avoiding some context switching overhead. The Peregrine *local* RPC implementation differs specifically from LRPC in two additional areas. First, LRPC was implemented only on VAX processors, which provide a separate *argument pointer* register in the hardware and procedure calling convention. By using the argument buffer as the stack for the new server thread, we avoid the need for a dedicated argument pointer register. Also, unlike LRPC, the argument buffer in Peregrine is never accessible in both the client and server address spaces at the same time, avoiding the problem of allowing other threads in the client to modify the arguments during the call.

The V-System kernel is a message-passing system [5]. As a result, the optimizations specific to RPC are not available to the V kernel. Gather DMA for packet header and data is used in some implementations. The arguments are copied into the server's address space. For multi-packet arguments, the copy is done on-the-fly as each packet arrives, such that only the last packet adds to the latency. The optimistic blast protocol implementation [4] attempts to avoid these copies by predicting that the next packet to arrive during a blast is indeed the next packet in the blast protocol transfer. Packet headers are derived from a header template.

In Sprite [12], RPC is used for kernel-to-kernel communication, thereby avoiding concerns relating to copies between user and kernel space. Sprite makes limited use of gather DMA for transmitting the arguments and return values. The gather capability of the network interface is used to transmit packets consisting of a header and a data segment, usually a file block, which need not be contiguous. The data of multi-packet arguments or return values are copied on-the-fly, as in the V kernel. The data segment consists of bytes, while the header is made up of "words." The header contains a tag, indicating to the receiver whether data conversion is necessary. Header templates are used to avoid recomputation. A pool of kernel daemon threads handles incoming calls; the stack and the registers of these threads are saved and restored.

The x-kernel is a testbed for modular yet efficient implementation of protocols [7]. The x-kernel's RPC implementation is composed of a number of "smaller" protocols [8]. Arguments to an RPC call are put into a single contiguous buffer by the client stub. Small arguments are copied into the server's address space, while a transparent optimistic blast protocol implementation attempts to avoid copies for large arguments [11]. Data is transmitted in the native format of the sender, with a flag indicating the sender's architecture. The receiver checks this flag, and performs the necessary conversion, if any. Each of the component protocols use a template for its protocol header, which is then copied into a header buffer. RPCs are executed by a server thread selected from a pre-allocated pool of threads. The thread persists across invocations; its registers and stack are saved and restored.

The Firefly RPC system [15] was implemented on a small shared-memory multiprocessor. Copies are avoided in a variety of ways. Arguments are put into a single buffer in user space, and transmitted from there using DMA. On the server side, a collection of packet buffers is statically mapped into all address spaces, and thus available to the server without further copies. Header templates are built at bind time, but the UDP checksum is completely recomputed for every packet. A pool of server threads accepts incoming calls; the threads maintain their user stack and registers between calls, but the kernel stack is discarded. The data representation can be negotiated at bind time.

Amoeba is a message-passing kernel [19]. RPC is built on top of the kernel's synchronous message-passing primitives by stubs that can be generated automatically or by hand. As with the V kernel, the optimizations specific to RPC are not available to the Amoeba kernel. Before transmission by the kernel, the stub copies all argument or result values into a single contiguous buffer. Amoeba does not enforce any data representation conversion of these buffers; any needed conversion is left to the stub writer. Within the kernel, Amoeba completely rebuilds the packet header for each transmission, without the use of header templates.

SunRPC is implemented with a collection of library routines on top of the Unix kernel [17], and thus the optimizations specific to RPC are not available to SunRPC. The stubs copy all arguments or result values into a single contiguous buffer, converting them to a standard data representation [16], before requesting the kernel to transmit them as a message. As a special case in some implementations, if the client or server transmitting an RPC message uses the same native representation as the network standard representation for some data types, representation conversion is bypassed for those values. Some versions of the Unix kernel use header templates for transmitting consecutive network packets on the same connection.

In terms of performance comparison for RPC and message-passing systems, two metrics seem to be common: the latency of a null RPC (or message) and the maximum data throughput provided by a series of RPCs with large arguments (or a series of large messages). Table 7 shows the published performance figures for network communication in a number of RPC systems and message-passing operating systems. The machine name and type of processor used for each system are shown, along with an estimate of the CPU speed of each in MIPS [15]. Except where noted, all performance figures in Table 7 were measured between two user-level threads executing on separate machines, connected by a 10 megabit per second Ethernet.

8 Conclusion

We have described the implementation and performance of the Peregrine RPC system. Peregrine supports the full generality and functionality of the RPC model. It provides RPC performance that is very close to the *hardware* latency, both for *network* RPCs, between two user-level threads

System	Machine	Processor	MIPS	Latency (microsec.)	Throughput (Mbits/sec.)
Cedar [3]*	Dorado	Custom	4	1097	2.0
Amoeba [19]	Sun-3/60	MC68020	3	1100	6.4
x-kernel [8]	Sun-3/75	MC68020	2	1730	7.1
V-System [5]	Sun-3/75	MC68020	2	2540	4.4
Firefly [15]	5-CPU Firefly	MicroVax II	5	2660	4.6
Sprite [12]†	Sun-3/75	MC68020	2	2800	5.7
Firefly [15]	1-CPU Firefly	MicroVax II	1	4800	2.5
SunRPC [17]‡	Sun-3/60	MC68020	3	6700	2.7
Peregrine	Sun-3/60	MC68020	3	573	8.9

*Measured on a 3 megabit per second Ethernet.

†Measured kernel-to-kernel, rather than between two user-level threads.

‡Measurements reported by Tanenbaum et al. [19].

Table 7 Performance comparison of network RPC and message-passing systems executing on separate machines, and for *local* RPCs, between two user-level threads executing on the same machine. Peregrine has been implemented on a network of Sun-3/60 workstations, connected by a 10 megabit per second Ethernet. In the Peregrine system, the measured latency for a null RPC over the network is 573 microseconds, which is only 309 microseconds above the *hardware* latency for transmitting the required packets. For large multi-packet RPC calls, the network user-level data transfer rate reaches 8.9 megabits per second, over the 10 megabit per second Ethernet. Between two user-level threads on the same machine, the measured latency for a null RPC is 149 microseconds.

We have described the benefits of various optimizations that we used in the implementation of Peregrine. In particular, we avoid copies by transmitting arguments and return values directly from user space, and by mapping the arguments into the server's address space. We have found these optimizations to be beneficial even for the smallest argument and return value sizes. Further savings were obtained by using header templates and avoiding recomputation of the header on each call, by avoiding data representation conversions for communication between machines with identical native data representations, and by reducing thread management overhead on the server.

Acknowledgements

Mike Burrows, Fred Douglis, Frans Kaashoek, Sean O'Malley, John Ousterhout, Larry Peterson, Mike Schroeder, and Brent Welch provided insights into the design of the RPC systems in which they participated. We thank them for their help, and we hope that we have represented their systems accurately in our comparisons with their work. We also wish to thank John Carter, Alan Cox, Mootaz Elnozahy, Pete Keleher, and the referees for their comments on the paper.

References

[1] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[2] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[3] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] J.B. Carter and W. Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '89)*, pages 61–69, May 1989.

[5] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[6] D.R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[7] N.C. Hutchinson and L.L. Peterson. The x-kernel: An architecture for implementing protocols. *IEEE Transactions on Software Engineering*, SE-17(1):64–76, January 1991.

[8] N.C. Hutchinson, L.L. Peterson, M.B. Abbott, and S. O'Malley. RPC in the x-kernel: evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.

[9] G. Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[10] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, May 1981.

[11] S.W. O'Malley, M.B. Abbott, N.C. Hutchinson, and L.L. Peterson. A transparent blast facility. *Journal of Internetworking*, 1(2):57–75, December 1990.

[12] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[13] J.B. Postel. Internet Protocol. Internet Request For Comments RFC 791, September 1981.

[14] ROSS Technology, Inc., Cypress Semiconductor Company. *SPARC RISC User's Guide*, second edition, February 1990.

[15] M.D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[16] Sun Microsystems, Inc. XDR: External data representation standard. Internet Request For Comments RFC 1014, Internet Network Working Group, June 1987.

[17] Sun Microsystems, Inc. *Network Programming*, May 1988.

[18] Sun Microsystems, Inc. The SPARC architecture manual, version 8, January 1991.

[19] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[20] W. Zwaenepoel. Protocols for large data transfers over local networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32, September 1985.