The Peregrine High-Performance RPC System

David B. Johnson
Willy Zwaenepoel

Rice COMP TR91-151
March 1991

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

(713) 527-4834
dbj@rice.edu, willy@rice.edu

Submitted for publication

This work was supported in part by the National Science Foundation under grant CDA-8619893.

Abstract

The common usage patterns and required semantics of the remote procedure call (RPC) model provide many opportunities for improving the performance of RPC. We explore the three observations that most RPCs do not require data representation conversion for arguments or results, that most network RPCs occur between machines on the same local network, and that the semantics of RPC do not require server threads to retain any thread-specific state between calls. To demonstrate the optimizations possible based on these observations, we have implemented a new RPC system, called Peregrine[1]. We concentrate primarily on optimizing *network* RPC performance, between two threads on separate machines, but also support *local* RPC, between two threads executing on the same machine.

The Peregrine system has been implemented on a network of SUN-3/60 workstations connected by a 10 megabit per second Ethernet. In this implementation, a null RPC between two user-level threads executing on separate machines over the network requires only 562 microseconds. This time compares well with the fastest existing null network RPC times reported in the literature, ranging from about 1100 to 2600 microseconds, and is only 298 microseconds above the *minimum* possible *hardware* latency for transmitting the call and result packets in our hardware environment. For larger 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 only 149 microseconds.

[1]The peregrine falcon, the world's fastest flying creature, can reach speeds of over 200 miles per hour.

1 Introduction

The common usage patterns and required semantics of the *remote procedure call (RPC)* model provide many opportunities for improving the performance of RPC. In the RPC model, servers export one or more interfaces, making a specific set of server procedures available to clients that bind to the corresponding interface. Clients then invoke these procedures in the servers by performing RPCs, which appear to the client as a normal procedure call and to the server as a normal procedure invocation. In reality, the client actually invokes a client stub procedure that marshals the call arguments and the identification of the procedure to be invoked into a call message, which is delivered to the server machine. There, 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.

In systems using RPC, the total size of the arguments or results of a call is generally small. Bershad et al. found that the majority of RPCs transfer fewer than 200 bytes [Bershad 90a]. Similarly, Schroeder and Burrows report that more than 95 percent of the RPCs that occur in their system require only one Ethernet packet for the arguments or results [Schroeder 90]. This common usage allows protocols for network RPC to concentrate on the single-packet case, and allows simpler allocation and management of buffer space, even for local RPCs that do not go over the network. A similar observation was made earlier by Birrell and Nelson, who optimized Cedar RPC [Birrell 84] for the case in which only a single packet was required. Bershad et al. [Bershad 90a] also found that complex argument and result marshaling was seldom required, and that simple byte copying was sufficient for transferring arguments and results on most RPCs. Since the semantics of RPC require the client to bind to the server interface before calls can take place, control and data structures can be preallocated and initialized, saving the expense of rebuilding them on each call [Schroeder 90, Bershad 90a].

In this paper, we explore three specific RPC usage and semantics observations and the opportunities for RPC performance optimization that they present. We concentrate primarily on optimizing *network* RPC performance, between two threads on separate machines, but also support *local* RPC, between two threads executing on the same machine. High-performance network RPC is important to the performance of distributed application programs executing in parallel on multiple machines on a network, cooperating in the solution of a single problem. To demonstrate these optimizations, we have implemented a new RPC system, called Peregrine, that provides RPC performance that is very close to the minimum possible *hardware* latency. For *network* RPCs, this minimum latency is bounded by the expense of sending the call and result message over the network. This expense, which is termed the *network penalty* [Cheriton 83], is the hardware cost for transmitting a packet of a given size over the network from one machine to another, and is measured without operating system or interrupt latency. It differs from the *network transmission time* for packets of the same size, because it includes the required processor and network interface costs for sending and receiving the packets, in addition to the network transmission time. For *local* RPCs, the minimum RPC latency is bounded by the expense of the required procedure call, kernel trap handling, and context switching overheads [Bershad 90a]. Although we have optimized for the semantics defined by RPC, we have not sacrificed these semantics for the sake of performance, and have maintained the same semantics as conventional RPC implementations [Birrell 84].

The Peregrine RPC system has been implemented 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 latency for a null RPC between two user-level threads executing on separate SUN-3/60 workstations on the Ethernet is only 562 microseconds. This time compares well with the fastest existing null network RPC times reported in the literature, ranging from about 1100 to 2600 microseconds, and is only 298 microseconds above the *minimum* possible *hardware* latency defined by the network penalty for the call and result packets in our hardware environment. With a single 1-kilobyte argument, a null RPC requires only 1364 microseconds, showing an increase over the time for null RPC of just the network transmission time for the additional bytes of the call packet. This time is 305 microseconds above the network penalty, and is equivalent to a user-level data transfer rate of 6.01 megabits per second. For larger multi-packet RPC calls, the network user-level data transfer rate reaches 8.90 megabits per second, coming within 95 percent of the bandwidth limited by the network penalty. The implementation of Peregrine uses the thread and memory management facilities of the V-System [Cheriton 83], but is not specific to the V-System and does *not* use the V-System's `Send-Receive-Reply` message passing primitives or protocols.

In Section 2 of this paper, we describe the three specific RPC usage and semantics observations that we explore to develop our optimizations. Section 3 provides an overview of the Peregrine RPC system, including the environment in which it has been implemented. Section 4 describes the implementation of these optimizations in the Peregrine system, including the implementation of single-packet network RPCs, multi-packet network RPCs, and local RPCs. An evaluation of the performance of the Peregrine system and the effectiveness of these optimizations is presented in Section 5. Section 6 compares our work to other RPC systems, and finally, Section 7 presents conclusions and discusses possible further work.

2 Optimizing RPC Performance

In this paper, we examine three observations on the common usage patterns and required semantics of RPC systems, and consider the optimization of RPC performance based on those optimizations. Specifically, we examine the following three RPC observations:

Observation 1: Most RPCs do not require data representation conversion for arguments or results when passing between the client and server address spaces [Bershad 90a]. Rather, the arguments can be used verbatim in the server's address space, and the result values can likewise be used verbatim in the client's address space.

Observation 2: Most network RPCs occur between machines on the same local network, rather than requiring forwarding through gateways [Shoch 80].

Observation 3: The semantics of RPC do not require server threads to retain any thread-specific state (i.e., registers or stack contents) between calls [Birrell 84]. The semantics require only that the specified procedure is executed by some thread in the server's address space; these threads need not continue to execute, or even to exist, between calls.

Based on these observations, we have derived a number of optimizations that can be applied to an RPC implementation. These optimizations have been implemented in the Peregrine RPC

system, and the performance reported in Section 5 demonstrates the opportunities afforded by these observations. RPCs in the Peregrine system run at very close to the minimum possible processor and network *hardware* latency in our environment, for both network and local RPCs. The optimizations implemented in Peregrine include the following:

- For network RPCs, the call arguments and results are sent directly from the user's address space, without marshaling. This is possible because in the common case, both client and server machine use the same data representation and because most values are of simple data types.

- Because the client must bind to the exported server interface before RPCs can occur, we are able to preallocate server threads and to build call and result packet header templates before they are needed. This greatly reduces the per-call overhead.

- Since the RPC semantics do not require the server to retain thread-specific state between calls, we can effectively create a new server thread to handle each call. Rather than actually creating a new thread, though, we utilize a pool of preallocated server threads and simply reinitialize one for each new call, requiring only a few fields in the thread descriptor to be modified. Using a "new" thread for each call allows avoids much of the context switching and kernel trap overhead when starting the thread on the call and later when the return trap is executed, since no registers need be saved or restored. By preallocating these threads at bind time, we are able to get the benefits of dynamically creating a new thread for each call without the expense of a full thread creation operation.

- Also, because we are able to dynamically create a "new" server thread for each call, the server kernel's network receive packet buffer can simply be mapped into any convenient location in the server's address space, and can then be used as the new thread's stack, with the call arguments on the top of the stack. This avoids the cost of copying the arguments into the server's address space. If, instead, an existing thread with thread-specific state were used, the thread would already have a stack and would likely have registers pointing to locations on that stack. When a call packet arrives, the waiting thread would generally also have already specified to the kernel the address at which the next RPC call packet should be placed. By not retaining thread-specific state between calls, we are free to place the stack at any convenient address and to start the thread with the packet data on the top of its stack.

- Since most calls have small arguments and return values, we optimize for the case in which all arguments fit in a single call packet and all return values fit in a single result packet.

3 Overview of the Peregrine RPC System

The Peregrine RPC system follows the conventional RPC model of servers exporting one or more interfaces, and clients binding to an interface before calling any of the procedures defined in that interface [Birrell 84]. As in Firefly RPC [Schroeder 90], the choice of transport protocol and protocol implementation for RPCs between a particular client and server pair is postponed until binding time, allowing the system to use the protocol and implementation appropriate to this binding. For a client and server pair executing on separate machines over the network, the system may choose

between using an RPC protocol layered directly on the local network data link layer when client and server agree on this optimized approach and are both connected to the same local network, or using an RPC protocol layered on a standard internetwork protocol such as TCP/IP. Between a client and server pair executing on the same machine, the system may also choose shared memory for argument and result passing. Binding initializes data structures in the kernel for use on each call and return, and returns a *binding handle* to the client program. The binding handle provides a form of capability in the kernel, similar to a Unix file descriptor [Bach 86], ensuring that no RPCs can be executed without first binding. In addition to the binding handle, a word of *binding flags* is returned, indicating to the client the particular protocol and implementation to be used, and identifying any different data representation used by the server machine for which the client must marshal arguments and unmarshal results.

The system presents a conventional procedure call interface to the client and server, using stub procedures. No special programming is required in the client or server, and no compiler assistance is required. Argument values are pushed onto the call stack by the client as if calling the server routine directly. The binding handle 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 stubs, the associated binding handle 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 tested on each call. Between a client and server pair both using Peregrine on the same type of machine, the entire binding flags word is zero; if the flags word is nonzero, separate code is used to determine individually which flag is set and to handle the indicated special processing for this binding.

Procedure arguments may be immediate data values, or may be pointers to buffers of arbitrary size. Pointer arguments may optionally be declared in one of three ways, depending on whether the buffer must be passed on the call, the return, or both [Schroeder 90]:

- An *in* pointer argument describes a buffer that is to be passed on the call, but which need not be passed back on the return.
- An *out* pointer argument describes a buffer that must be included on the return, but which need not be passed on the call.
- An *in-out* pointer argument describes a buffer that must be passed on both call and return.

If not specified, pointer arguments are assumed to be *in-out*. There may be any number of arguments, but there is currently a limit of six pointer arguments, due to the kernel's configuration of the Ethernet interface.

In the Peregrine implementation, additional traps were added to the kernel to support RPC call and return, and minor changes were made in several areas of existing 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 of first-level trap and interrupt handling (before calling a C procedure) and context switching support.

4 Implementation

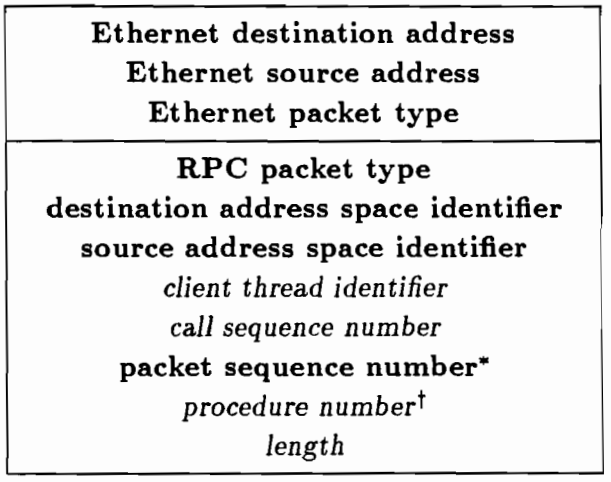
In the Peregrine implementation, the kernel is responsible for getting RPC packets from one address space to another, and for starting a new thread in the server or unblocking the client thread

when the corresponding packet arrives. The kernel also retransmits packets and is responsible for acknowledgements and duplicate packet detection. The network protocol used is similar to Birrell and Nelson's protocol in Cedar RPC [Birrell 84]. All processing specific to the particular server procedure called is performed in the stubs, simplifying the kernel design. For example, the stub describes to the kernel any buffers that must be included in the packet, but the knowledge of *which* arguments are pointers to these buffers and the conversion of those pointers between address spaces is handled by the stubs.

4.1 The Packet Header

The packet headers of the call packets sent from a particular client to a particular server change little from one RPC to the next. The same property also applies to the headers of the result packets sent by that server back to the same client. Many of the fields of the RPC header, and of the enclosing network header, can be determined in advance at bind time. Figure 1 shows the combined Ethernet and RPC header for the Peregrine RPC protocol, when layered directly on the Ethernet data link protocol. All fields in the packet header denoted in boldface can be determined at bind time and do not change until the client or the server terminates. Only the client thread identifier, the call sequence number, the procedure number, and the argument or result length change from one call or result packet to the next.

At bind time, the client's kernel allocates a number of *call packet header templates* equal to the expected maximum number of concurrent calls from this client to this server for the interface being imported. This number may be specified by the client on the bind request or will be defaulted by the kernel. Similarly, the server's kernel allocates the same number of *result packet header templates.* In the current Peregrine implementation on a network of uniprocessor machines, the default number



*The packet sequence number indicates the location of this packet in the argument or result data. For single-packet arguments or results, it is always 0.
†Only for call packets.

Figure 1 RPC and Ethernet packet header

of each type of header template is 1, since although address spaces are multithreaded, we expect in the common case for there to be only one outstanding call between a particular client and server. The fields of the packet header indicated in boldface in Figure 1 are initialized in the packet header templates at bind time. The other fields are filled in when a packet is sent.

Call packet header templates are deallocated when the client address space terminates. Result packet header templates are deallocated on termination of the server address space or when the server becomes aware of the termination of the corresponding client address space. A server may also deallocate result packet header templates at other times, for instance when it runs out of space for templates or after a prolonged period of inactivity from the corresponding client address space. When a packet needs to be transmitted without a header template being present, a new template is allocated and initialized for this packet, and is retained for future packets.

4.2 Single-Packet Network RPC

4.2.1 Sending the Call Packet

When a process performs an RPC, the arguments are pushed onto the its stack and a procedure call to the client stub is performed as if the actual server procedure were being called directly. The client stub pushes the length of the argument list and the procedure number onto the stack, loads the binding handle into a register, and traps into the kernel. The argument list length and procedure number are passed to the kernel on the stack rather than in registers since they must be included in the call packet, and this places them in memory with the arguments already on the stack. The binding handle need not be passed on the stack since it is not included in the call packet. The stack also contains the client's procedure return address, pushed by the procedure call to the stub. The stack contents at the time of the trap is illustrated in Figure 2.

For any pointer arguments, code exists in the stub to handle each in one of three ways, depending on whether the buffer must be passed on the call, the return, or both. Before the kernel trap on the call, the client stub pushes onto the stack a descriptor for each *in* or *in-out* pointer argument, giving the address and length of the referent buffer. When the packet is transmitted, any *in* buffers are added to the call data after the header and arguments, followed by any *in-out* buffers. The descriptors are necessary since the kernel does not know which arguments are pointers and does not otherwise know the length of each buffer. The data of these buffers is transmitted directly from the client's address space, without copying. Any *out* pointer arguments are treated as immediate data values in transmitting the call packet, and do not use buffer descriptors on the call.

The LANCE Ethernet interface used in our implementation has a *gather* DMA capability, common of many modern network interfaces. Given a list of segments, each specified by a start address and a length, the LANCE can transmit a single Ethernet packet consisting of these segments, without copying them into a common contiguous packet buffer. We use this capability to transmit the packet from the packet header template built at bind time, the arguments on the client's stack, and the buffers for any *in* and *in-out* pointer arguments. However, I/O using DMA on the SUN can proceed only to or from virtual addresses in kernel memory. Therefore, for each of these segments other than the packet header template (which already resides in kernel memory), the virtual memory pages in which they reside are first double-mapped into kernel space before instructing the LANCE to transmit the packet. The client's procedure return address is included in the packet

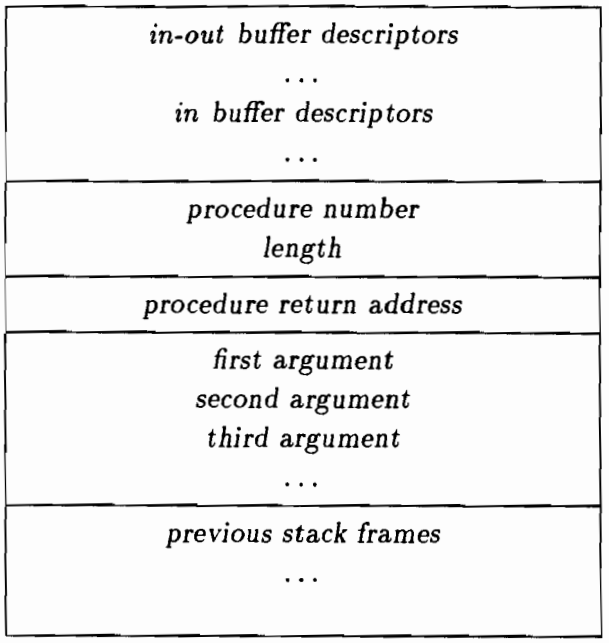


Figure 2 Client stack at entry to the kernel

only because it appears on the client's stack between the call arguments pushed by the user, and the procedure number and length pushed by the stub; it is ignored by the server.

After transmitting the packet, the calling thread is blocked awaiting the return from the server. If no response is received from the server, the packet is retransmitted in the same way as the original packet. Although retransmitting these segments from the client's address space allows them potentially to be modified by other threads in that address space between retransmissions, we view this as more a problem of the client providing insufficient synchronization of its own threads, than a problem of the RPC system. This problem could be avoided by copy-on-write protecting these segments [Fitzgerald 86], but we do not currently do so.

4.2.2 Receiving the Call Packet

The requested procedure is called by the server stub as if it had been called directly as a local procedure call by the client. For the server stub to make this procedure call, the arguments for the call must first be on the server thread's stack, as if they had been pushed by the client. To avoid copying the arguments from the received packet buffer to the stack, we arrange to use the packet buffer itself as the server thread's stack. At initialization time, the Ethernet interface is configured with a number of receive buffers, each with size equal to the maximum Ethernet packet size, such that each buffer is at the top (high-address end) of a separate virtual memory page. Although this requires more memory to hold the receive buffers, the additional expense is negligible. Once received, the layout of the call packet in this page is as shown in Figure 3.

On arrival of a packet, the Ethernet interrupt routine examines the packet. If it is an RPC call packet, the interrupt routine passes control to the RPC implementation, which first determines

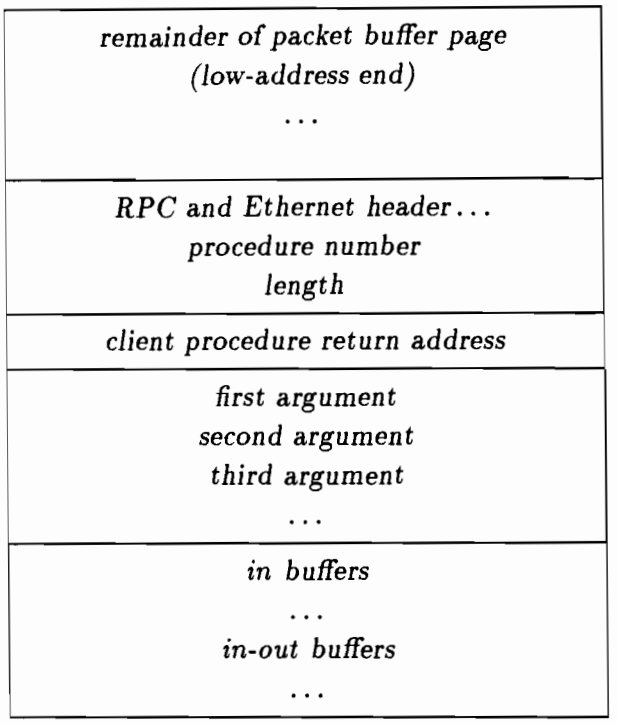


Figure 3 Received call packet in Ethernet buffer page

whether it is a duplicate, using the call sequence number and the calling thread identifier and a corresponding table in the kernel. The receive packet buffer page is then transformed (without copying) into a stack for the server thread to handle this call: the page is remapped into the server's address space, and a new unused page is remapped to the original virtual address of the packet buffer to replace it for future packets. Since the stack on the Motorola MC680x0 grows downward, placing the Ethernet receive buffer at the top of this page leaves the remainder of the page for use as stack space for the server thread. Additional stack pages may also be allocated below this Ethernet receive buffer page to provide additional stack space, if specified by the server when the interface was exported. Once the Ethernet buffer has been remapped into the server's address space, a preallocated server thread is reinitialized, or a new thread is created if necessary. The thread begins execution at the server's RPC dispatch procedure registered at export time, with the arguments and procedure number on the top of its 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. For any pointer arguments, though, that argument, which now contains a pointer in the client address space, must first be overwritten with a corresponding pointer in the server address space. For *in* and *in-out* pointers, the address of the buffer now on the server stack (from the packet) is used. For *out* pointers, though, no buffer was sent in the packet, and a new buffer is instead created and the *out* pointer on the stack for this

argument is overwritten with the address of this new buffer. The stub then calls the corresponding user procedure in the server.

4.2.3 Sending the Result Packet

When the server procedure returns to the stub, the functional return value of the procedure is returned in register `d0` (and `d1` if floating point), following the MC680x0 compiler convention. If there are no *in-out* or *out* pointer arguments, the stub immediately traps into the kernel with `d0` and `d1` as returned by the server procedure. If there are *in-out* or *out* pointer arguments, though, a descriptor for each referent buffer is first pushed onto the stack, specifying to the kernel the address and length of each separate buffer that must be passed back in the result packet. Once in the kernel, the result packet is transmitted using the LANCE Ethernet gather DMA interface, in the same way as the call packet was transmitted (Section 4.2.1). Any result buffers are then 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 then unprotected. As with the Cedar RPC protocol [Birrell 84], 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. Instead, it is added to the pool of available server threads for a future call packet. When reused for the next call, the thread is reinitialized and retains no thread-specific state (i.e., no registers or stack contents) from this call. 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. This is also what makes possible the use of the Ethernet receive packet buffer as the server thread's stack (Section 4.2.2). By not retaining thread-specific state between calls, the stack can be placed at any convenient address, and the server thread can simply be started with the packet data on the top of its stack. If instead an existing thread were used, it would already have a stack, with registers pointing to locations on that stack, and would also have already specified to the kernel the address at which the next RPC call packet should be placed.

4.2.4 Receiving the Result Packet

When the result packet is received by the client kernel, the functional return value is copied into the client thread's saved registers. If there are no *in-out* or *out* buffers being returned, the client thread is then immediately unblocked and resumed, completing the call. However, if there are buffers included in the results, they are first copied from the packet buffer into the client's address space. When the client thread resumes, it begins executing again 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 slight modification to conventional RPC semantics. Pointers to *out* or *in-out* buffers would need to be passed 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 [Sun 88] to avoid the expense of copying, and a similar mechanism has been

implemented in Accent [Fitzgerald86] and Mach [Accetta86] for message passing. We have not currently implemented this mechanism in Peregrine because we want to preserve conventional RPC semantics.

4.3 Multi-Packet Network RPC

For a network RPC in which the argument or result message is larger than the data portion of a single Ethernet RPC packet, the message must be broken down into multiple packets for transmission over the network. 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 4.2.

To transmit the arguments of an RPC in multiple packets, a sequence of packets each of maximum size is transmitted. As in the single-packet case, the data is transmitted directly from the client's address space using gather DMA to avoid copying. The packet header for each is transmitted from the packet header template built at bind time. For each packet, the *packet sequence number* (Figure 1) is incremented to indicate the location of this packet in the argument message for this call. The data is transmitted in the same order as for a single-packet call, except that the packet that would naturally be sent *first* is instead sent *last*. That is, the first data packet transmitted does not contain the beginning of the arguments, but rather starts at an offset into the argument data equal to the maximum data size of a packet. Following this, the remainder of the packets are sent in their natural order, followed finally by what normally would have been the first packet. This unusual order of transmitting the packets of the argument data is used to avoid latency caused by copying the data at the server, as explained below.

On receipt at the server machine, these packets are assembled into a single contiguous buffer. For all but the last packet received (the naturally first packet), the packet data is copied into a contiguous argument buffer. The copying begins on a page boundary for this first packet received. By transmitting the first packet last, we are able to overlap the overhead of these copies with the transmission of the following packet. When the last packet arrives, it is instead remapped into a page located immediately before the assembled argument buffer. As described in Section 4.2.2, the Ethernet receive buffers are each located at the top (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 this call, a contiguous argument buffer is assembled *without copying* the data of this final packet. The copying of the data of all but the last packet is overlapped with the transmission of the following packet, but this is not possible for the last packet transmitted, so we instead avoid the copy overhead by arranging to be able to remap this last packet buffer to complete the arguments. Since the Ethernet receive buffers are each at the top of a separate page (with unused space below it in the same page), only the naturally first packet of the call data can be remapped in this way to assemble a contiguous buffer containing the arguments. Since the arguments of this call are larger than a single packet, using remapping for this first packet coincidentally also guarantees that we avoid copying what must be a completely full packet.

When the argument buffer is complete, it is remapped into the server's address space and becomes the new server thread's stack, in the same way as is done for single-packet RPCs. The thread begins execution in the server dispatch procedure, with the arguments on the top of its stack, and the *in* or *in-out* buffers on its stack above the arguments. The server dispatch procedure

branches to the specific server stub code for the specified user procedure, which then calls the correct user procedure. When the call returns to the stub, the stub traps into the kernel, and the results are transmitted back to the client machine. If the results fit into a single packet, they are transmitted and the client thread is restarted as described in Section 4.2.3. If the results require more than a single packet to transmit, they are transmitted as described in this section for the multi-packet call data, but the client thread is restarted as for the single-packet case.

4.4 Local RPC

The implementation of local RPC in Peregrine is similar to the method used in LRPC [Bershad 90a]. LRPC provides local RPC performance that is substantially faster than that of conventional RPC implementations. However, LRPC was only implemented on the DEC Firefly using VAX processors, which provide a separate *argument pointer* register in the hardware and procedure calling convention. LRPC uses a buffer mapped into memory shared by both client and server to pass arguments and result values, and need only set the argument pointer register to the address of this buffer in the server to allow the server thread access to its arguments. This technique, though, cannot be used directly on other processors, such as the Motorola MC680x0, which lack an argument pointer register.

Instead, we use an argument passing technique similar to our remapping of the Ethernet packet receive buffer for network RPCs, to form the server thread's stack (Section 4.2.2). Like LRPC, the arguments are copied in the client stub from the stack to a separate buffer used for communication with the server. However, the kernel remaps this buffer from the client's address space to the server's address space on the call, forming the server thread's call stack in the same way as is used for the Ethernet receive buffer. The client thread is switched into the server's address space, rather than creating a new server thread, and its stack pointer is initialized as if it had just begun execution with this buffer as its stack. Switching the client thread's address space generally involves only a change in its virtual memory context register. The client thread thus becomes the server thread for this call, and begins execution again in the server dispatch procedure, as for network RPC. On return from the call, the stack buffer is remapped back into the client thread's address space, the server thread again becomes the client thread, and its stack pointer is restored to point to its original stack. Upon leaving the kernel on the RPC return, execution resumes in the client stub, which then returns from the procedure call to the calling user.

Unlike in LRPC, the argument buffer is never accessible in both the client and server address spaces. Rather, it is remapped between address spaces, and is only mapped into the address space in which the client/server thread is currently executing. This prevents other threads in the client's address space from potentially modifying the arguments during the call, and avoids the extra argument copies necessary with LRPC to provide immutable arguments. Although this requires two extra memory remapping operations over LRPC (to remap the buffer on the call and return), this expense is negligible for the improved semantics of making all arguments immutable.

5 Performance

All performance measurements presented in this paper were obtained using diskless SUN-3/60 workstations, each with a 20-megahertz Motorola MC68020 processor. Each workstation is connected

to a 10 megabit per second Ethernet network using an AMD Am7990 LANCE Ethernet network controller. The network was otherwise idle during measurement in order to avoid variance caused by network traffic from other workstations. 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 then divided by the number of iterations. Among individual trials for the same measurement, most results either remained constant or varied by less than 1 microsecond per iteration. For each measurement, the performance figures presented have been averaged over several individual trials.

5.1 The Network Penalty

The *network penalty* [Cheriton 83] for sending the RPC call and result packets represents the minimum possible cost for network RPCs. The network penalty is the time to transfer a given number of bytes over the network from one machine to another on an idle network, assuming no errors. It is a function of the processor, the network, the network interface, and the number of bytes transferred. The transfers are implemented at the network data link level, busy waiting for receipt of each packet, and thus no protocol overhead, context switching, or interrupt processing costs appear in the result. The network penalty is obtained by measuring the transfer time and dividing the result by 2. It thus represents the cost of transfering data in one direction only; any acknowledgements required by particular higher-level protocols incur a separate network penalty. Table 1 shows the measured network penalty for single-packet transfers in our hardware environment, for various packet sizes ranging from the minimum to the maximum Ethernet packet size. The transmission time shown is the theoretical time to transfer the packet over a 10 megabit per second Ethernet (0.8 microseconds per byte), whereas the network penalty time is as measured in our environment. The additional 18 bytes in each packet consist of the Ethernet destination address, source address, packet type, and CRC; no overhead has been added to allow for various protocol headers other than the required Ethernet header. The overhead shown represents the difference between the network penalty and the theoretical transmission time.

For data transfers larger than the maximum Ethernet packet data size, the network penalty measures the minimum cost of streaming the required number of packets over the network. The

Table 1

Network penalty for various packet sizes

Data Size (bytes)	Total Size (bytes)	Transmission Time (microseconds)	Network Penalty (microseconds)	Overhead (microseconds)
46	64	51.2	132	80.8
100	118	94.4	173	78.6
200	218	174.4	255	80.6
400	418	334.4	417	82.6
800	818	654.4	743	88.6
1024	1042	833.6	927	93.4
1500	1518	1214.4	1314	99.6

packets are sent as quickly as possible, with no delay between packets and no protocol acknowledgements. Table 2 shows the measured network penalty for multi-packet data transfers in our hardware environment, for various multiples of the maximum Ethernet packet data size. The bandwidth shown represents the network bandwidth utilized by each data transfer. Due to the additional preamble and synch bits added to the packet in transmission, and due to the required interpacket gap time for back-to-back transmission enforced by the LANCE Ethernet interface, data transfers cannot utilize the full 10 megabits per second bandwidth of the Ethernet [Carter89].

5.2 Single-Packet Network RPC

The performance of the Peregrine RPC system for single-packet RPCs is very close to the network penalty times given in Table 1. Table 3 summarizes our measured RPC performance and compares it to these theoretical minimum times. The network penalty 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 time 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 breakdown of the total overhead, each component of 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 occuring in the client before transmitting the call packet, those occuring in the server, and those occurring in the client after receiving the result packet. This breakdown illustrates the effect of several of the optimizations we have made based on the RPC usage and semantics observations discussed in Section 2.

The cost of remapping the Ethernet receive buffer in the server to become the new server thread's stack is 4 microseconds. In comparison, the kernel's general-purpose memory copying routine requires 12 microseconds to copy only 4 bytes (and 9 microseconds to copy 0 bytes). Thus, even for a null RPC, remapping the stack is more efficient than calling the general-purpose memory

Table 2

Network penalty for multi-packet data transfers

Data Size (bytes)	Number of Packets	Network Penalty (milliseconds)	Data Bandwidth (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 3

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

Procedure	Network Penalty	Measured	RPC Overhead
Null RPC	264	562	298
4-byte argument RPC	264	563	299
1024-byte *in* RPC	1059	1364	305
1024-byte *in-out* RPC	1854	2321	467

copying routine. This routine, though, has a high startup overhead, since it determines different special cases for loop unrolling during the copy. For copying small argument lists, the correct number of assignment statements could be executed in-line, but the overhead of determining this special case would be larger than the small cost of simply remapping the buffer. 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, for copying 1024 bytes, the kernel's memory copying routine requires 159 microseconds.

Table 4

Measured breakdown of costs for a null network RPC (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 sending call packet	49
Network penalty for call packet	132
Server Ethernet receive interrupt handling	43
Duplicate packet detection	5
Page remapping of receive buffer into server stack	4
Other server handling of call packet	8
Server context switch and kernel trap on return	26
Server stub	4
Server sending result packet	16
Network penalty for result packet	132
Client Ethernet receive interrupt handling	43
Client handling of result packet	5
Total measured cost breakdown	511
Measured cost of complete null RPC	562

The use of preallocated and initialized packet headers for both the client and server threads can be seen in the cost of transmitting the call and result packets. For transmitting the call packet, the cost is 49 microseconds, and for transmitting the result packet, the cost is 16 microseconds. Both of these are substantially lower than the cost of allocating a new packet header buffer and initializing it for each packet sent. In our implementation, the cost of sending the call packet is more expensive than that of sending the result packet due to the need to first double-map the pages of the client's stack into kernel memory. As mentioned in Section 4.2.1, the Sun hardware can perform DMA I/O only to or from addresses in kernel memory. In the case of the null RPC call packet, the procedure number and argument list length are transmitted from the client's stack. For the null RPC result packet, this expense is not present since there is no data from the server's stack to be transmitted back to the client. As in the case of remapping the Ethernet receive buffer into the server's stack, the cost of remapping the data from the client's address space is less than the general cost of copying the data to the packet header buffer.

Since the server thread has no state between calls, its registers are not saved or restored when it begins execution at the start of a call or when it ends execution when it sends the results. The savings achieved by this optimization are evident in the differing kernel trap and context switch overhead of the client and the server threads. In our implementation, this optimization saves 11 microseconds per null RPC. On processors with larger numbers of registers that must be saved and restored on a context switch and a kernel trap, this optimization will increase in significance [Anderson 90].

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, Ethernet transmit interrupt handling for the call packet, blocking the calling thread, and the context switch to the next thread to run while waiting for the results. On the server's machine, the operations that occur in parallel with other components of the RPC cost breakdown include saving the return values for possible later retransmission, Ethernet transmit interrupt handling for the result packet, and the context switch to the next thread.

5.3 Multi-Packet Network RPC

Like the single-packet network RPC performance, the performance of multi-packet network RPC is bounded by the network penalty of sending the call and request messages, which must be divided into separate packets. The measured multi-packet performance achieved by Peregrine is very close to this minimum network hardware cost. 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 buffers are transmitted through the system, and does not include the size of the Ethernet or RPC packet headers. For large transfers, Peregrine achieves a throughput of up to 8.9 megabits per second, coming within 95 percent of the bandwidth limited by the network penalty. When reassembling multi-packet arguments or results, the cost of copying all but the last packet is overlapped with the transmission cost of the following packet. In our implementation, copying the data of a packet requires 209 microseconds, while the network penalty for each packet is 1314 microseconds. Thus, this copying requires only about 16 percent of the minimum time between packets.

Table 5

Peregrine RPC performance for multi-packet network RPCs

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	43.31	8.87
48000-byte *in-out* RPC	81.66	86.27	8.90

5.4 Local RPC

Unlike network RPC performance, the performance of local RPC is bounded mainly by processor overheads [Bershad 90a]. In particular, the theoretical minimum cost of a local null RPC is the sum of the following required hardware costs:

- a procedure call from the client user program to the client stub,
- a kernel trap and change of virtual memory address space on the call, and
- a kernel trap and change of address space again on the return.

The kernel trap and address space switch costs depend heavily on the processor architecture and operating system design [Anderson 90]. The figures reported for these operations in Table 4 apply to the local RPC case as well, making the theoretical minimum null local RPC cost in our environment 65 microseconds.

Table 6 summarizes the measured performance of local RPC in our implementation. The Peregrine local null RPC time of 149 microseconds compares well with the reported null RPC time for LRPC of 157 microseconds [Bershad 90a], although the SUN-3/60 used in our measurements has two advantages over the C-VAX processor used in LRPC: the processor is faster, and the Sun memory management hardware does not require a TLB flush on a context switch. For LRPC, Bershad et al. cite a performance loss of approximately 25 percent due only to the TLB misses caused by this flush.

Table 6

Performance of local Peregrine RPC (microseconds)

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

Relative to the theoretical 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, 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 Comparison to Other Systems

Our approach to *network* RPC is similar in some respects to optimizations used by the LRPC system [Bershad 90a], but LRPC is designed only for *local* RPC, between two threads 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. For example, LRPC preallocates a set of *argument stacks* at bind time for use in passing arguments in shared memory to the server. This preallocation is roughly similar to our preallocation of packet header templates at bind time. LRPC also exploits the same observation that we make, that server threads need not retain thread-specific state between calls, but applies this observation only to the case of local RPCs. LRPC uses the client thread directly in the server's address space to execute local RPCs. While we also use this approach for local RPCs, we exploit the semantics of server threads for network RPCs as well. The URPC system [Bershad 90b] also uses much the same mechanism as LRPC, but URPC is designed only for use between threads executing on the same shared memory multiprocessor machine.

The Firefly RPC system [Schroeder 90] exploits the binding required by RPC semantics to select the appropriate transport mechanism for calls between the client and server. The system can choose between transport over the network by a custom protocol encapsulated in UDP Internet packets, network transport by the DECNet byte stream protocol, or transport to another thread on the same machine through shared memory. The Peregrine system allows a similar choice at binding time, using either our network RPC protocol, or our local RPC transport through the argument buffer remapped between client and server. We likewise allow the choice of any standard Internet protocol or other heterogeneous RPC protocol through the use of the binding flags returned at binding time. At call time, the client stub simply tests the binding flags, and if nonzero, branches to code that determines which flag bit was set and executes the appropriate code for that alternate protocol. For each such alternate protocol to be made available, code is provided in the Peregrine client stub to branch to a standard client stub for the protocol being used.

Exploiting the binding required by RPC semantics has also been explored in another context by Bershad et al. in the HRPC system [Bershad 87]. At binding time, clients learn the particular RPC protocol and data representation used by the server, and select the appropriate implementation for use on each call with this server. However, their work was concerned more with interoperation with various heterogeneous native RPC protocols than with performance enhancement.

A number of message-passing distributed operating systems [Tanenbaum 90, Cheriton 83] use a request-response protocol similar to that of RPC systems. However, since server threads in message-passing systems must retain thread-specific state between a reply and the next request, these systems cannot utilize the types of optimizations based on the corresponding semantics of RPC systems. Many such message-passing systems, though, exploit the observation that most network communication occurs between machines on the same local network. For example, the Amoeba sys-

tem [Tanenbaum 90] uses a network protocol layered directly on the Ethernet data link layer for most communication. To allow gatewaying Amoeba communication beyond the local network, Amoeba provides a wide-area network gateway service [vanRenesse 87]. The V-System [Cheriton 83] takes a similar approach, by utilizing a specialized network protocol at the Ethernet data link layer for most communication, but providing for communication beyond the local network through standard Internet protocols implemented by a server process on the local network. We follow this approach for communication on the local network, but instead use the binding flags to allow the client to directly utilize a more general Internet protocol for communication when the server is located beyond the local network. In the context of other RPC systems, research with the x-kernel [Hutchinson 89] has also explored the use of transport protocols layered directly on the Ethernet for communication within a local network, and dynamically adding Internet protocol layers when necessary for servers not on the local network.

Performance comparison of different RPC systems is difficult, due to differing processor speeds, operating system overheads, and network interface latencies. Two performance parameters seem to be common in comparisons between systems, though: the latency of a null RPC and the maximum data throughput provided by a series of RPCs with large arguments. Table 7 shows the published performance figures for network communication in a number of RPC systems and message-passing operating systems, adapted from Schroeder and Burrows [Schroeder 90]. 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.

Table 7

Performance comparison of network RPC and message-passing systems

System	Machine	Processor	~MIPs	Latency (microsec.)	Throughput (Mbits/sec.)
Cedar [Birrell 84]*	Dorado	Custom	1 × 4	1097	2.0
Amoeba [Tanenbaum 90]	SUN-3/60	MC68020	1 × 3	1100	6.4
x-kernel [Hutchinson 89]	SUN-3/75	MC68020	1 × 2	1730	7.1
V-System [Cheriton 88]	SUN-3/75	MC68020	1 × 2	2540	4.4
Firefly [Schroeder 90]	FF	MicroVax II	5 × 1	2660	4.6
Sprite [Ousterhout 88]†	SUN-3/75	MC68020	1 × 2	2800	5.7
Firefly [Schroeder 90]	FF	MicroVax II	1 × 1	4800	2.5
SunRPC [Sun 88]‡	SUN-3/60	MC68020	1 × 3	6700	2.7
Peregrine	SUN-3/60	MC68020	1 × 3	562	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. [Tanenbaum 90].

7 Conclusions and Further Work

We have explored three observations on the common usage patterns and required semantics of RPC, and the opportunities for RPC performance improvements that they present. First, most RPCs do not require data representation conversion for arguments or results when passing between client and server address spaces [Bershad 90a]. In addition, most network RPCs occur between machines on the same local network [Shoch 80]. Finally, the semantics of RPC do not require server threads to retain any thread-specific state (i.e., registers or stack contents) between calls [Birrell 84].

By exploiting these three observations, we have developed a number optimizations for RPC systems, and have demonstrated these optimizations through the implementation of the Peregrine RPC system. Peregrine provides RPC performance that is very close to the minimum possible *hardware* latency, both for *network* RPCs, between two user-level threads on separate machines, and for *local* RPCs, between two user-level threads executing on the same machine. The Peregrine RPC system 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 only 562 microseconds, which is 306 microseconds above the minimum network *hardware* cost. For large multi-packet RPC calls, the network user-level data transfer rate reaches 8.90 megabits per second. Between two threads on the same machine, the measured latency for a null RPC is 149 microseconds.

In addition to further optimizing our current implementation based on additional RPC usage and semantics observations, we are currently expanding the system to provide interoperability with several other existing RPC protocols, such as SunRPC [Sun 88]. We have completed a port of Sun's RPC code to our system but have not yet integrated it with the Peregrine client stubs. The testing of the binding flags to determine if an alternate protocol and client stub are to be used on each call is already in place, though. We are also examining the possible effects on our optimizations of future advances in processor or network hardware. In particular, we are interested in the effects of using network hardware with much higher bandwidth, such as 100 megabit per second FDDI. For example, our current implementation copies the data of multi-packet RPCs in order to assemble a single contiguous argument or result buffer, but new techniques may be required with higher speed networks. To address this, we are currently considering an optimistic implementation of the protocol on the receiving side [Carter 89], to allow the data to be assembled contiguously without copying.

References

[Accetta 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the Summer Usenix Conference*, July 1986.

[Anderson 90] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. Technical Report 90-08-01, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, August 1990.

[Bach 86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Bershad 87] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[Bershad 90a] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[Bershad 90b] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. Technical Report 90-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, July 1990.

[Birrell 84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Carter 89] John B. Carter and Willy Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In *1989 ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems: Proceedings*, pages 61–69. ACM, May 1989.

[Cheriton 83] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[Cheriton 88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Fitzgerald 86] Robert Fitzgerald and Richard F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[Hutchinson 89] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. RPC in the *x*-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 91–101. ACM, December 1989.

[Ousterhout 88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.

[Schroeder 90] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Shoch 80] John F. Shoch and Jon A. Hupp. Measured performance of an Ethernet local network. *Communications of the ACM*, 23(12):711–721, December 1980.

[Sun 88] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification, version 2. Request For Comments RFC 1057, Internet Network Working Group, June 1988.

[Tanenbaum 90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[vanRenesse 87] Robbert van Renesse, Andrew S. Tannenbaum, Hans van Staveren, and Jane Hall. Connecting RPC-based distributed systems using wide-area networks. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 28–34. IEEE Computer Society, September 1987.