System Support for Efficient Network Communication

Chandramohan A. Thekkath

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 94-07-02
July 17, 1994

This technical report is an adaptation of C. Thekkath's Ph.D. dissertation. Our research program is supported in part by the National Science Foundation under Grants No. CCR-8703049, CCR-8619663, and CCR-8907666, CDA-9123308, and CCR-9200832, the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer. C. Thekkath was supported in part by a fellowship from Intel Corporation.

University of Washington

Abstract

System Support for Efficient Network Communication

by Chandramohan A. Thekkath

Co-Chairpersons of Supervisory Committee: Professor Henry M. Levy
Professor Edward D. Lazowska
Department of Computer Science
and Engineering

Recent advances in processor and network technology have significantly changed the hardware base for distributed systems. However, high-speed networks and fast processors alone are not sufficient to deliver the end-to-end performance that users of these systems expect; good software design is critical. This thesis explores software mechanisms needed to support efficient network access for the next generation of workstations and networks.

The thesis proposes a new communication mechanism or model based on remote access (read and write) to protected memory segments. This model has two key aspects. First, it simplifies input data demultiplexing and reduces data copying costs. This allows users of the model very efficient access to remote data. A second and more significant feature of the model is that it separates the transfer of data from the transfer of control. That is, a remote data access can complete at the target process without invoking a thread of control within the target.

The communication model is implemented on DECstation workstations connected by an ATM network and provides performance close to the limits imposed by the hardware. Using example applications, we show how the model can provide excellent performance for both parallel applications and conventional distributed system mechanisms like remote procedure call (RPC).

However, we argue that conventional communication mechanisms, e.g., message passing or RPC, do not fully exploit the potential of newer networks. These mechanisms closely couple the transfer of both control and data. We demonstrate, using the read/write com-

munication model, that separating data transfer and control transfer in distributed systems can both (1) eliminate unnecessary and expensive control transfers and (2) facilitate tighter coupling of a distributed system. This has the potential to increase performance and reduce server load, which supports scaling in the face of an increasing number of clients.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

It is a pleasure to acknowledge my advisors, Hank Levy and Ed Lazowska, for their exceptional counsel throughout my graduate career at the University of Washington. I would like to especially thank Hank ("Zen Master") Levy for asking me difficult questions that caused me to reexamine many of my ideas about my work. I owe an enormous debt to Ed Lazowska for his insights and advice, both technical and otherwise, and for his constant support through many difficult phases of graduate school. Between the two of them, they taught me how to do research and be a teacher of computer science. It has been an absolute privilege to be associated with them.

I would like to thank both of them and Larry Ruzzo for serving on my reading committee.

My thesis research has benefited from conversations with several people including: Tom Anderson, I. Ashok, Brian Bershad, Jeff Chase, Ed Felten, Hal Murray, Chuck Thacker, John Wilkes, and John Zahorjan.

I would like to thank the DEC Systems Research Center for inviting me to spend a summer there as an intern, and Mike Burrows for hosting me. DEC SRC also supplied me with generous amounts of equipment. Thanks are due to Eric Cooper of FORE Systems Inc. for loaning me an ATM switch for my experiments. Thanks also to Intel Corporation for supporting me with a Fellowship.

I would also like to thank my fellow students, especially my office mates in 433, for making my stay at UW such a wonderful learning experience.

I am thankful to Alicen Smith and Frankye Jones for making sure that I did not run afoul of the graduate school administration.

Any system building activity, such as that described in this thesis, requires a great deal of supporting infrastructure. I would like to thank Nancy Johnson Burr and Jan Sanislo for their help with this.

On a more personal note, I would like to thank my family, especially my mother, for their complete and unquestioned support of my dreams. Finally, it is with a deep sense of gratitude that I thank Radhika Thekkath; I would not have been able to survive graduate school without her love and understanding.

Chapter 1
INTRODUCTION

A cluster of workstations connected by a local-area network has traditionally been used as a hardware base for distributed systems. Such a configuration, with appropriate software, has long held the promise for significant cost-performance, flexibility, and scalability benefits over more dedicated architectures. However, relatively slow communication links and high software processing overheads can impose a stiff penalty for cross-machine communication. These high costs can limit the extent to which a workstation cluster can be used as a solution for either structuring-related or performance-related problems in distributed applications.

Some recent developments, however, offer the potential to change the way workstation clusters can be used:

- New switch-based network technologies, such as Asynchronous Transfer Mode (ATM), offer more than a 10-fold bandwidth improvement over Ethernet [46]. Another order-of-magnitude improvement to one gigabit per second seems close at hand. Further, these switch-based networks are scalable since multiple network links can be aggregated to give bandwidths far greater than individual link bandwidths.

- New processor technologies and RISC architectures have already given us an order of magnitude improvement in processing power, with further improvements in the offing. Workstations using these processors are capable of using the bandwidth that new networks provide. Also, processors in modern high-end workstations are typically comparable to the fastest available uniprocessors and superior to those used in massively parallel machines.

Taken together, these technology advances represent a dramatic change in distributed system hardware. This change, in turn, should enable us to build a new generation of novel, high-performance distributed applications. Consider the following two motivating examples.

In a cluster of networked workstations, paging to spare memory lying idle on other nodes could be much faster than paging to disk [20]. A distributed server could manage the idle memory in the network; on a page fault, the page fault handler would communicate with the server and transfer pages to or from remote nodes. The effectiveness of this scheme is highly influenced by the communication performance between client and server.

As another example, better communication would greatly facilitate the use of networked workstations as a loosely-coupled multiprocessor or shared virtual memory system [14, 41].

Careful software design is essential if we are to exploit the advances in hardware. High-speed networks and fast processors are necessary but not sufficient by themselves. We illustrate this point below with a simple example.

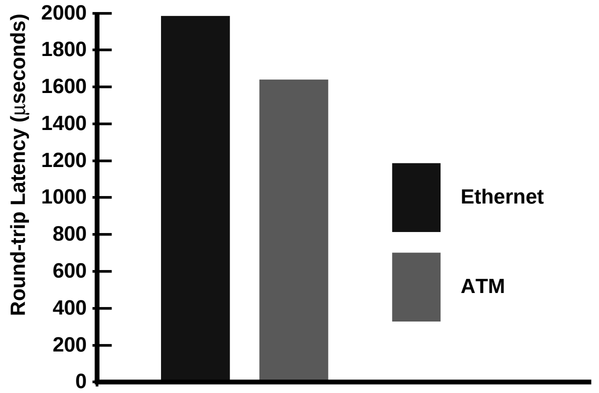


Figure 1.1: Performance of Ultrix Remote Procedure Call

Distributed systems are typically structured using remote procedure call (RPC) as the cross-machine communication mechanism [12, 51]. Figure 1.1 shows the performance of the Ultrix remote procedure call system between a pair of otherwise idle 25 MHz DECstations making small packet RPCs. In one case, the machines are connected by the traditional 10 Mb/sec Ethernet, and in the other, by a modern 140 Mb/sec ATM network. The RPC system being measured is in everyday use as the underlying communication mechanism for distributed services on Ultrix and other Unix-like systems. Ultrix RPC software has not been particularly optimized and thus, on a network that is over ten times faster, the performance of small packet exchanges does not change significantly. Unfortunately, small data exchange is a common occurrence in distributed systems [6]. In short, the advantages gained from increased network hardware performance can be almost completely masked without proper software design.

Designing software mechanisms so that applications can efficiently use the next generation of high-speed networks is a difficult challenge. This thesis offers new software solutions to improve the performance and the usefulness of distributed systems by exploiting opportunities enabled by technology. New technology has created two distinct opportunities for software innovation.

First, by suitable design we should be able to build low-latency and high-bandwidth communication mechanisms on the new networks. This, in turn, enables us to improve the performance of distributed systems that rely on these low-level mechanisms. In other words, we seek improved system performance by improvements in traditional low-level communication primitives, such as RPC, upon which conventional distributed systems are built.

Technology trends also create a second, and far more significant, opportunity for new software design, especially for switch-based, local-area networks. While the hardware base for distributed systems is changing dramatically, both in terms of speed and reliability, the software *structure* of distributed systems has not changed significantly in recent times. Rather than redesigning just the lower levels of the communication system, we can reevaluate our basic notions of distributed systems structure, making it possible to build systems with performance more commensurate with hardware capabilities.

The next two sections set out, in greater detail, the challenges in building high-performance distributed systems, how current designs address them, and what the new contributions of this thesis are.

1.1 The Challenges for High-Performance Distributed Systems

In a distributed system, good application level performance is fundamentally dependent on the performance of four layers of the system as sketched in Figure 1.2.

These layers can be thought of as a vertical slice through a distributed system. They are *not* supposed to indicate the various layers of the ISO OSI reference model, but simply the various parts of a distributed system discussed in this thesis. The layers are interdependent,

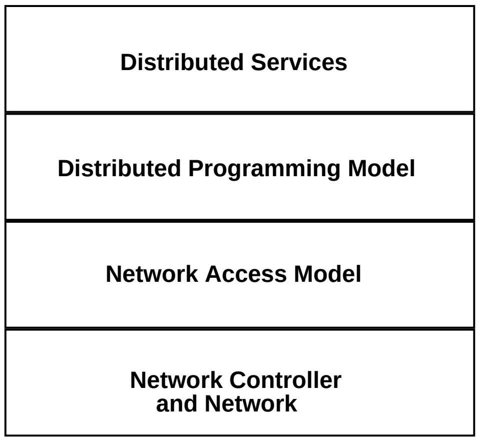


Figure 1.2: Layers of a Distributed System

i.e., a particular layer provides services to the layer above it and uses the services of the layer below it.

The uppermost layer of the system represents *distributed services*. Familiar examples of distributed services are distributed file systems such as NFS [56], name servers such as the Internet Domain Name Service DNS, electronic mail services, and others.

Distributed services are programmed using various *distributed programming models*. Perhaps the most common example of a distributed programming model is remote procedure call (RPC). Variations of RPC include message passing as seen in the V system [16] and CSP [33]. Another example of a distributed programming model is distributed shared memory (DSM) such as Ivy [41].

Distributed programming models are implemented on top of a lower level abstraction referred to, in this thesis, as a *network access model*. An example of a network access model is the familiar byte-stream abstraction to the network. For instance, Unix-like systems provide the notion of sockets that applications can use to send and receive streams of data bytes. Distributed programming models, e.g., RPC in Unix, are layered on top of this byte-stream abstraction. Other distributed programming models such as distributed shared memory can also be implemented on top of the byte-stream abstraction [41]. It is also important to note that similar abstractions, albeit with different performance or guarantees, might be implemented at different levels. For instance, using a network access model based on unreliable byte-streams, one might provide a reliable, byte-stream based distributed programming model to applications.

Finally, network access models are layered on top of the *network controller* that interfaces the processor to the network. Some well-known examples of controllers include the LANCE [1], the VMP-NAB [38], and the Nectar CAB [5]. Typically, controllers buffer network packets, provide checksum and encryption, and have sophisticated data transfer capabilities between the network and host memory (such as scatter-write and gather-read). Some even have on-board implementations of entire protocol stacks.

In recent years, technology has improved the performance of the network. However, in current systems, the performance seen at the distributed services layer has not benefited significantly from the advances in technology. There are two closely related reasons for this.

The first reason is that there are mismatches among the various layers of the distributed system. A mismatch between two layers can lead to performance inefficiencies that can accrue to degrade overall performance. For example, complexities of the controller affect the performance of the network access model. Similarly, if the network access model provides an interface that is somehow incompatible with typical distributed programming models implemented on top of it, performance can suffer.

In a related vein, a distributed programming model that either does not exploit the characteristics of new networks or is overly restrictive can limit the performance of distributed services that use it. Distributed programming models, such as RPC and many of the structuring principles embodied in current systems, date back to the time of 0.5 MIPS Xerox Alto workstations interconnected by the 3 Mb/sec experimental Ethernet [65]. For instance, traditional distributed systems have been structured around the client/server model, in which clients access services using message passing or RPC. This model has the advantage of providing a convenient and familiar programming abstraction to users. However, the RPC model encourages an isolation between client and server that may limit performance as technology permits tighter coupling. Performance may also be compromised by complexities inherent in RPC-style communication.

1.2 The Scope and Contribution of this Thesis

This thesis discusses techniques for enhancing end-user communication performance through improvements in the various layers mentioned above.

First, it considers the design of controllers and the effect of controller design on the overall performance of the distributed system. In the past, controller design has concentrated on throughput; latency has been largely ignored. However, due to the request-response nature of common distributed programming models like RPC, latency and not just throughput is often crucial to system performance. We mentioned earlier that one approach to improving overall distributed systems performance is to improve the performance of distributed programming models. Chapter 2 studies the limitations imposed on this approach to improving performance.

Next, the thesis considers an alternative network access model that allows the efficient implementation of a variety of distributed programming models. It has good performance and shows little mismatch with a variety of programming models and can be efficiently implemented. In fact, the performance of RPC implemented on the new network access model is superior to a comparable implementation using a traditional byte-stream based model on identical hardware.

However, next generation local-area networks have many characteristics that make it appropriate to *directly* use the memory-based network access model as the distributed programming model, in preference to alternatives like RPC. Fundamentally, distributed programming models such as RPC involve control and data transfer, both of which can be expensive. By directly using the memory-based network access model, it is possible to restructure distributed systems for higher performance by avoiding control transfer and by using an optimized data transfer mechanism instead.

To summarize, this thesis makes the following contributions.

First, it identifies ways of improving controller design and makes the observation that using the appropriate network access model can significantly improve the performance of communication primitives.

Second, it proposes a network access model based on remote memory. The thesis demonstrates the effectiveness of the model in getting higher performance in diverse situations such as RPC and workstation-based multicomputing. The model proposed here goes beyond earlier memory-based approaches, such as that proposed by Spector [61], by incorporating notions of virtual memory, protected sharing of the network, and separate data and control transfer. Separating data and control transfer allows function shipping mechanisms (e.g., RPC) and pure data shipping mechanisms to be optimized separately.

Third, this thesis proposes a way of structuring distributed systems to achieve high performance by exploiting the separation of control and data transfer mechanisms provided by the network access model.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 evaluates the impact of network controller designs on communication performance. Chapters 3 and 4 are central to the thesis and describe the network access model. Following these are two chapters describing experience with layering distributed programming models over the network access model. Chapter 5 describes a prototype RPC implementation and demonstrates that using the memory-based network access model gives very high performance. Chapter 6 describes the organization of a cluster of workstations as a multicomputer using the network access model as the underlying basis for communication. Chapter 7 describes an alternative organization of distributed systems based on the network access model and evaluates its performance implications. Chapter 8 puts the work described in this thesis in the context of related research. Chapter 9 offers conclusions and some avenues for future research.

Chapter 2

THE IMPACT OF NETWORK CONTROLLERS ON COMMUNICATION PERFORMANCE

As mentioned in the previous chapter, a network controller interfaces the host processor and memory system to the network. Typical network controllers for workstation-class machines are designed to be located on the I/O bus along with other peripherals, such as disk controllers. Depending on the controller, it can support a wide variety of functions to offload the processing of network data from the host. Typically, controllers provide support for checksumming and data transfer mechanisms, e.g., direct memory access (DMA) between the network and host memory. In some cases, controllers have embedded processors that can be used to implement entire transport protocols [5, 38].

In this chapter we evaluate the overhead added to cross-machine communication from two important sources: (1) the network controller hardware and (2) low level controller software, memory, and CPU interfaces. Given different network and processor technologies, the relative importance of these components may change; by isolating the components, we can see how the performance of each scales with technology change.

Controllers targeted for high throughput have been well studied in the past [5, 24, 38, 68]. However, latency is often the overriding concern in distributed systems, and consequently, for the most part, this chapter considers the impact of the controller on this aspect of communication performance.

Given the increases in processor and network speed, it is natural to hypothesize that low latency communication is easily achievable on newer workstation clusters. For example, through software design and increased processor speeds, it should be possible to reduce the software protocol processing time. In fact, even early studies of the Internet TCP protocol determined that protocol processing per se was not an insurmountable source of latency [17]. Further, with the increased network speed, it should be possible to rapidly transfer data between the source and destination machines.

However, this hypothesis is too simplistic because it ignores an important component of potential cross-machine communication overhead—the network controller. That is, good protocol software, fast processors, and high network speeds notwithstanding, network controllers that are poorly matched to the demands of modern distributed systems can lead to bad performance.

There are two conflicting goals that are often traded off in the design of controllers for high-speed networks. One goal is not to burden the host processor with frequent interrupts as network packets arrive. As an extreme example, on a 155 Mbit/sec ATM network, individual *cells*, which are fixed size 53 byte packets, can arrive at the host once every 2.7 microseconds. In order to isolate the host from frequent interrupts and protocol processing overheads, some designs migrate functionality into the controller [24, 68]. However, the migration of too much functionality into the controller can impact the latency of processing network messages. In addition, if the controller presents a complex interface to the host, there could be additional software latency introduced by the host device driver in managing the controller.

There are two potential sources of controller-related communication inefficiencies that we should distinguish between. First, the *internal* design of the network controller introduces inherent latency into the communication path. For example, a complex hardware and software architecture on-board the controller can increase latency. Second, the *external* interface to the host determines the latency costs in managing the controller. For example, depending on the interface, initiating a transmission or dismissing a device interrupt on receives could introduce excessive latency. As another example, the particular data transfer mechanism supported by the controller might not be well suited to the host processor and memory architecture. This mismatch could also cause latency problems.

In this chapter, we study the effect of both internal and external controller interfaces on communication latency. Our methodology is to make careful measurements to analyze the impact of various controller and host combinations. The chapter is organized as follows. We briefly characterize the experimental testbed and describe the measurement techniques. Then, we analyze the performance results and discuss the issues for controller design that affect latency in cross-machine communication.

2.1 Overview of the Networking Environment

Our networking environment consists of traditional Ethernets as well as the faster FDDI and ATM networks. Each network is accessed using a different type of controller. The networks are driven using DECstation 5000/200s and SparcStation I workstations.

2.1.1 Ethernet

The Ethernet is a 10 Mbit/sec CSMA/CD local-area network, which is accessed on the DECstations and SparcStations by a LANCE controller [1]. The controller is packaged differently on the two machines. On the DECstations, the controller cannot do DMA to or from host memory; instead, it contains a 128 Kbyte on-board packet buffer memory into which the host places a correctly formatted packet for transmission [27]. Similarly, the controller places incoming packets in its buffer memory for the host to copy. In the case of the SparcStation, the controller transfers data to and from host memory using DMA. In this case, a cache flush operation is done on receives to remove old data from the cache. On both machines, packets are described by special descriptors initialized by either the host (on send) or the controller (on receive). Descriptors are kept in host memory on the SparcStations while they are in the special on-board packet memory on the DECstations. Two message sizes were used in our experiments, a minimum sized (60 bytes) send and receive, and a maximum sized (1514 bytes) send and receive.

2.1.2 FDDI

FDDI is a 100 Mbit/sec fiber token ring, accessed on the DECstations by the DEC (DEFZA) FDDI controller. Like the DECstation Ethernet controller, the FDDI controller cannot perform DMA from host memory on message transmission; instead, it relies on an on-board packet buffer memory. However, the FDDI controller can perform DMA transfers directly to host memory on reception of a packet from the network. The controller and host software share descriptors as described above for the DECstation Ethernet. We used an unloaded private FDDI ring with two hosts. Thus, the overhead due to token passing is kept to an absolute minimum; in a more realistic environment token passing delay would have to be added to the overall latency. Packet sizes of 60 and 1514 bytes were chosen to facilitate direct comparison with Ethernet.

2.1.3 ATM

ATM (Asynchronous Transfer Mode) is an international telecommunication standard used to implement B-ISDN. In an ATM network, data is exchanged between entities in fixed length parcels called cells, usually on the order of a few tens of bytes. An ATM network typically consists of a set of hosts connected by a collection of switches that form the network. In an ATM network, user-level data is segmented into cells, routed, and then reassembled at the destination using header information contained in the cells.

The particular ATM used has 140 Mbit/sec fiber optic links and cell sizes of 53 bytes, and is accessed using FORE Systems' ATM controller [30]. The controllers on the two DECstation hosts were directly connected without going through a switch; thus there is no switch delay. Unlike the Ethernet and FDDI controllers, the ATM controller uses two FIFOs, one for transmit and the other for receive. The controller has no DMA facilities. The host simply reads/writes complete ATM cells by accessing certain memory locations that correspond to the FIFOs. The host is notified via interrupt when cells arrive in the receive FIFO. The host has considerable flexibility in choosing how often it should be interrupted. Further, the controller does not provide any segmentation or reassembly of cells; that is the responsibility of the host software. Each ATM cell carries a payload of 44 bytes; in addition there are 9 bytes of ATM and segmentation-related headers and trailers. The experiments described below used packet sizes that were an integral number of cells as well as being close enough to the Ethernet and FDDI packet sizes for comparison.

2.2 The Testbed and Measurement Techniques

In order to isolate the performance of the controller and the network link, we built a minimal stand-alone software testbed, which simply sends and receives packets on the network. There is no operating system intervention except for the low level device driver and interrupt vectoring code. The testbed hardware consists of two workstations (either two DECstations or two SparcStations) connected through an isolated network. The DECstation uses a 25 MHz MIPS R3000 processor rated at 18.5 SPECmarks, and the SparcStation I uses a Sparc processor rated at 24.4 SPECmarks [60]. The DECstations were connected in turn to an Ethernet, an FDDI ring and an ATM network. The SparcStations were connected to an Ethernet. The DECstation network devices are connected on the 25 MHz TURBOChannel [26] while SparcStations use the 25 MHz SBus [64]. The performance of each configuration in sending a source packet from one node to the other and receiving a packet in response was measured. Measurements were averaged over at least 10,000 successful repetitions. Packets are sent and received from host memory, so the cost of moving the data over the host bus is included in the measurements. Network interrupts are enabled, so both the sending and the receiving hosts are interrupted on packet arrival. While it is generally possible to access the network in a dedicated fashion by disabling network interrupts, conventional time sharing access will involve interrupt processing overheads.

The component costs for the round-trip can be broken up as follows:

- **Time on the Wire:** This is the transmission time of the packet. The propagation time can be ignored because it is negligible for the length of cable connecting the two hosts.

- **Controller Latency:** This is the sum of two times: (1) the time taken by the sending controller to begin data transfer to the network once the host has made the data available to it, and (2) the delay between the arrival of the data at the receiving controller and the time it is available to the host.

- **Control/Data Transfer:** Data has to be moved at least once over the host bus between host memory and the network. Some of the controllers use DMA to transfer data over the host bus to the network; thus the CPU incurs no data transfer overhead. However, the CPU incurs a control transfer overhead because it has to use special memory descriptors to describe the location of the data to the controller. With such controllers, the actual time to do the data transfer is captured by the **Controller Latency** component.

- **Vectoring the Interrupt:** On the receive side, low level operating system software must vector the packet arrival interrupt to the interrupt handler in the device driver. The overhead involved is a function of the CPU architecture as well as the organization of the underlying system.

- **Interrupt Service:** On taking an interrupt, host software must perform some essential controller-specific bookkeeping before the interrupt can be dismissed.

To determine the latency of the controller itself, we ran separate experiments between a pair of hosts with interrupts disabled. Each host polled the controller's status registers in a loop. As soon as the register indicated arrival of data, a new transmission was begun. In the cases where the host was expected to copy data to the controller's memory before transmission, the host simply programmed the controller to start the data transfer, without actually giving it any data. Similarly, when the controller indicated the arrival of new data for the host to copy, the host ignored the data and began the next transfer. In addition, descriptors were prefilled before the data transfer started. In these circumstances very few machine instructions are executed by the host per round-trip. The time required to execute these as well as the time spent on the wire were subtracted from the total measured round-trip. This method gives satisfactory results in most cases but has the disadvantage that it does not account for any pipelining that the controller might perform. This artifact is particularly visible in the case of our ATM controller when multi-cell packets are exchanged. Typically a controller can overlap the transmission of data between its internal buffer and the network with the transmission between host (or on-board memory) and the chip itself. For instance, in sending a multi-celled packet through the ATM controller, the host can be filling the FIFO with a cell, while the controller is injecting the previous cell from the FIFO into the network.

2.3 Performance Analysis

Table 2.1 shows the cost of round-trip message exchanges on the host/network combinations described above. Before comparing the various combinations, a few points of clarification are in order here.

Table 2.1: Low-Level Round-Trip Packet Exchange Times

Component	**Round-Trip Time (μseconds)**							
	Packet Size in bytes (send/recv)							
	Ethernet (DEC)		**Ethernet (Sparc)**		**FDDI (DEC)**		**ATM (DEC)**	
	60/60	**1514/1514**	**60/60**	**1514/1514**	**60/60**	**1514/1514**	**53/53**	**1537/1537**
Time on the Wire	115	2442	115	2442	13	245	5	159
Controller Latency	51	53	89	103	97	230	16	161
Control/Data Transfer	40	600	6	6	40	253	17	458
Vectoring the Interrupt	25	25	12	12	25	25	25	25
Interrupt Service	26	26	42	42	92	140	9	20
Sum of Components	257	3146	264	2605	267	893	72	823
Measured Round Trip Time	253	3137	263	2611	263	894	73	746

First, in the case of the FDDI controller and the SparcStation controller, which perform DMA directly to host memory on packet receives, the cost of flushing the cache after the DMA is included in the the **Interrupt Service** overhead. While it is true that cache flushes are not strictly necessary if data from the network is kept in uncached memory locations, this means that the higher level software will eventually pay a performance cost of accessing this data.

Second, in the case of the ATM network, Table 2.1 does *not* include the cost for segmenting and reassembling multi-cell packets. In addition, the controller was programmed so that it interrupted the host only when *a complete packet* had arrived in the FIFO. Thus, in our experiments, each round-trip incurs only two interrupts.

Third, the measurements in Table 2.1 involve small amounts of system software written in a mixture of C and assembly language. Rewritting everything in assembly language would lead to a 1–2% (on the order of 10 microseconds) decrease in the round-trip latency of small packets.

Fourth, as noted earlier, both the DECStation and the SparcStation use the same Ethernet controller. However, recall that on the SparcStation the controller DMAs data over the host bus. The overhead for this is included in **Controller Latency**; **Control/Data Transfer** only includes the cost of the instructions to program the controller. The controller is able to overlap the data transfer over the bus with the transfer on the network. Thus the sum of **Controller Latency** and **Control/Data Transfer** is relatively unchanged by the packet size. In contrast, the DECstation controller incurs a heavy latency overhead because the host has to first copy the data over the bus before the controller can begin the data transfer.

The row **Sum of Components** is the sum of the rows above it. Most of the time, the sum of the component measurements is within 1–9 microseconds (about 2%) of the observed round-trip time. The only exception is in the case of the ATM network in sending multi-cell packets, where there is an overestimate of the round-trip time by about 12%. The most likely cause is an underestimate of the amount of overlap between the host memory–FIFO data transfers and the FIFO–network transfer.

The next several sections will consider the results of our study in detail. Overall, the high bandwidth of ATM and FDDI is significant for large packets. For example, latency for a 1514-byte packet on the DECstation Ethernet is 4.2 times worse than ATM and 3.5 times worse than FDDI. For packets even larger than 1514 bytes, the Ethernet situation is relatively worse, because FDDI will require fewer packet transmissions. The ATM comparison is slightly more complex; as pointed out earlier, ATM segmentation and reassembly costs were ignored in arriving at the figures in Table 2.1. The particular software implementation of the segmentation/reassembly in the driver requires about 11 microseconds per cell. If this is included, then a 1537 byte packet (29 cells) takes about 1065 microseconds, which is about 1.2 times the FDDI time. The situation further improves in favor of FDDI for packet sizes beyond this limit.

However, the packet-exchange times on Ethernet, FDDI and ATM show the important difference between network bandwidth and latency. If low latency for small packets is the goal, then a round-trip 60-byte message exchange on the DECstation Ethernet takes only 253 microseconds; FDDI on similar hardware, despite its 10-fold bandwidth advantage, takes 263 microseconds for the same operation, which is 4% *longer*. The increased FDDI bandwidth reduces the time spent on the wire, but other artifacts of the FDDI controller—increased latency and higher interrupt servicing time—result in overall increased latency.

2.4 Implications for Controller and Network Design

Based on the measurements just described, it is evident that higher speed networks do not necessarily imply proportionally lower latency. There are many artifacts of the controller and the processor and memory interface that could impact latency. We discuss these below, based on the experience with our controllers.

2.4.1 Impact of Internal Controller Structure on Latency

It is interesting to compare the ratio **Controller Latency** to **Time on the Wire** from Table 2.1. For small packets, this is 0.4 on the DECstation Ethernet, 0.8 on the SparcStation, 7 for the FDDI controller and 3.2 for the ATM network. For larger packets of approximately 1514 bytes, these ratios are respectively 0.02, 0.04, 0.9 and 1.0. While these numbers are specific to the controllers, they are indicative of a trend: *network speeds are improving dramatically, while controller latencies are not*.

One factor in controller latency is the complexity of the controller. The FDDI controller is the most complex of all our controllers and its latency is relatively higher than the others. It contains an on-board processor that executes a control program. In general, the control program is responsible for scheduling tasks within the controller. These tasks involve receiving incoming packets, managing internal buffers, interfacing with the host processor/memory subsystem, and others. An important aspect that influences communication latency is the type of scheduling that is done by the control program. Scheduling decisions made in a way that penalizes low latency could result in significant performance degradation. For instance, firmware revisions to the scheduling rules in the first version of the DEC FDDI controller led to a 22% reduction in latency for small packets at no cost to throughput.

2.4.2 Impact of External Controller Interface on Latency

There are two common types of interface that controllers export to the host to transfer data between memory and network. One interface designates a range (possibly all) of host memory to be used as a packet buffer and has the controller and the host share descriptors in memory. The alternative is to use a simple FIFO for transmits and receives, and have the host access it directly. Depending on the external interface supported by the controller, there are three overheads related to latency. First, there is the overhead of transferring data between the network and memory. Second, there is the interaction with the host memory hierarchy, and third, there is the cost of managing the controller and servicing its interrupts. We discuss each of these in turn.

Data Transfer Overheads

Our experiments, which included controllers that are capable of scatter-gather DMA, ordinary DMA and no DMA, allow us to examine some of the essential latency-impacting tradeoffs between these different controller types. Controllers that do not support DMA, such as our DEC Ethernet controller and the ATM controller, rely on programmed I/O (PIO) to move data. That is, the processor moves the data across the bus, usually without using a block transfer primitive.

In addition to the low-level details of data movement, e.g., DMA or PIO, two higher-level issues influence the cost of data transfer. These arise from protection and demultiplexing considerations because the communication system has to ensure that users cannot access unauthorized data. For example, on the sending side, the system has to ensure that the headers used to direct the packet on the network cannot be tampered with by the user. Similarly, incoming data has to be correctly demultiplexed before the data is copied to the authorized recipient.

Certain combinations of controller features and protection requirements make it difficult to restrict the number of data movements to one, the minimum achievable. Excessive and unnecessary data movement can lead to bad overall performance of the communication system.

Table 2.2 summarizes the number of copies that the controller, the kernel and the user need to perform, respectively, for each type of device: PIO, DMA and DMA with scatter-gather. The column **DMA S-G** represents scatter-gather DMA. The column **PIO (KM)** represents PIO with an optimization referred to as kernel-level marshaling.

Table 2.2: Number of Copies for Various Controller Types

Number of Copies (Device/Kernel/User)				
Fragment	**PIO (KM)**	**PIO**	**DMA**	**DMA (S-G)**
	Send			
Header	0/1/0	0/1/0	1/1/0	1/0/0
Data	0/1/0	0/1/1	1/0/1	1/0/0
	Receive			
Header	0/1/0	0/1/0	1/1/0	1/0/0
Data	0/1/0	0/1/1	1/0/1	1/0/0

Kernel-level marshaling differs from ordinary PIO in the following manner. In normal PIO, the user assembles pieces of non-contiguous data into a single buffer and traps into the kernel with a descriptor to the buffer. The kernel uses PIO to transfer this buffer into device memory. Thus each byte of user data is copied twice. In kernel-level marshaling, the user traps into the kernel with as many descriptors as there are non-contiguous data items. After verification, the kernel uses these descriptors to move data directly from user-space to the device. Thus, each byte is copied only once. Chapter 5 describes an RPC mechanism that uses this type of marshaling.

Certain implicit assumptions made in the table are clarified below. First, we are ignoring the cost of copies that might occur between the network and the controller's internal buffer. Typically this cost can be made negligible by using video RAMs or some similar technique. Second, we assume that with PIO, the on-board memory or FIFO cannot be reliably mapped into multiple user spaces, while with DMA, user data is mapped (instead of copied) to be adjacent to the header in kernel memory, so that the DMA can use a contiguous set of addresses. Finally, we assume that with scatter-gather DMA, the controller is capable of transferring arbitrarily small amounts of data over the bus and that on scatter DMA, the controller can perform address demultiplexing so that incoming data goes to the correct destination.

Ideally, one would like to minimize the number of times data is moved over the host bus between the network and the host memory. With PIO and kernel-level marshaling it is possible to keep the number of copies to one without compromising protection. With DMA this would not be possible, in general, because application level data could be located in multiple non-contiguous locations in memory. Most scatter-gather controllers (e.g., LANCE) have minimum size requirements for each segment and a maximum number of allowable segments. Thus, the user will have to marshal the data into one (or a few) location(s), and then have the controller move it. While it is possible to build controllers to overcome this restriction, using several small segments to gather data comes at a price, because the controller has to set up multiple DMA transfers, each of which incurs the cost of remastering the bus. A similar situation is true on the receiving side as well.

As shown in Table 2.2, using PIO with kernel-level marshaling allows the data copying cost to be kept to the minimum possible. However, one aspect of copying that is not captured in the table is the different rate at which data is moved over the bus for PIO and DMA. Typically, word-at-a-time PIO accesses over the bus are slower than block transfers using DMA; this is the case on both the DECstation TURBOChannel bus and the SparcStation SBus. While PIO versus DMA is of limited concern for short packets, there is a breakeven point beyond which PIO will be slower than DMA. Thus, unless the processor is required to touch each byte of the data for reasons other than moving it across the bus (for instance, to generate a checksum in software), it is usually more efficient to use DMA beyond a certain size.

Memory Hierarchy (Cache and TLB) Overheads

While Table 2.2 seems to indicate that PIO and scatter-gather DMA can perform the same number of copies, very often with current architectures the interaction of the memory subsystem with DMA might extract a heavier overall penalty than PIO.

In addition to the copying costs outlined above, without adequate support from the memory and processor subsystem, controller initiated data transfers could be a source of overhead due to cache effects. If the cache does not snoop on I/O operations, cache blocks could be left incoherent as a result of the DMA operation to memory, requiring cache flushes. If the cache is write-back, dirty entries may need to be purged before a DMA operation from memory.

Table 2.3 shows the contribution of the cache flush cost as a percentage of the total interrupt handling cost of the controllers with DMA. The total interrupt handling cost is the sum of cache flush costs and the essential controller-related bookkeeping overhead. The cache flush cost is simply the time taken to execute the instructions required by the host architecture to flush the cache lines corresponding to the data that was transferred. As is evident from Table 2.3, cache flushes are a serious penalty on current memory architectures. And in fact, the situation is even worse than the table suggests, because, in addition to the costs of executing additional cache flush instructions, cache flushes have a negative impact on performance by destroying locality. Newer processor/cache designs recognize this problem and provide memory coherence for DMA [59].

Table 2.3: Cache Flush Cost in Different Processor/Controller Configurations

	Received Packet Size in bytes			
	Ethernet (Sparc)		**FDDI (DEC)**	
	60	**1514**	**60**	**1514**
Total Interrupt Time	21	21	46	70
Cache Flush Time	3	10	7	30
Percentage Overhead	14	48	15	43

Another cost of DMA is the manipulation of page tables that is often necessary. On packet arrival, the controller stores the data on a page; however there is generally no way to guarantee that the page is mapped into the correct destination address space. Thus the kernel is faced with the option of either remapping or performing an extra copy.

To summarize the results of our experiments with the various controllers: DMA capabilities without adequate support from the cache and memory subsystem can be bad for performance in modern RISC processors. Further, controllers that have simpler interfaces to the host have the potential for reducing overheads. The next subsection argues for the use of simple controller/host interfaces.

Controller Management Overheads

Depending on the external interface, controllers can introduce varying amounts of software complexity on the sending and receiving sides. Based on our experience, ignoring the overheads of data transfer, on sends, there appears to be no significant cost difference in programming either a FIFO-based or a descriptor-based controller. However, there can be a significant difference in the overhead incurred in handling receive interrupts.

The experiments with the two types of controllers mentioned above indicate that the interrupt handling overhead can be significantly reduced by using a FIFO. Interrupt handling cost has two components: (1) the CPU-dependent cost of vectoring the interrupt and (2) the controller-dependent cost of servicing the interrupt. Previous research has studied interrupt vectoring costs on RISC processors [3, 55] and so we shall examine only the second component. The objective is to compare a simple FIFO interface such as that found in the ATM controller with a more elaborate descriptor interface such as that found in the Ethernet or the FDDI controllers.

Table 2.4 reproduces some of the measurements from Table 2.1 (recall that Table 2.1 shows round-trip times). It shows the cost of servicing the interrupt and transferring the data through a copy or a remapping operation, as appropriate. Since the intent here is to compare the interfaces and not the network, we have ignored the reassembly overhead on the ATM controller.

Table 2.4: Interrupt Handling Cost in Different Processor/Controller Configurations

	Transferred Packet Size in bytes							
	Ethernet (DEC)		**Ethernet (Sparc)**		**FDDI (DEC)**		**ATM (DEC)**	
	60	**1514**	**60**	**1514**	**60**	**1514**	**53**	**1537**
Interrupt Time	13	13	21	21	46	70	5	10
Copy/Mapin Time	11	201	9	39	10	10	5	138
Total	24	214	30	60	56	80	10	148

As the table indicates, for transferring small amounts of data to the user, the overhead of the FIFO-based controller (10 microseconds) is less than half that of the best-case descriptor-based controller (24 microseconds for the DEC Ethernet). For larger packets, the balance is in favor of controllers that allow the kernel to perform page mapping operations, because the copying cost dominates the interrupt handling cost. The ability to map data into user spaces at low cost is one alternative to kernel-level marshaling with PIO, which retains the benefits of protection and reduced data movement without the need to synthesize code. A typical limitation with FIFO-based controllers, such as the ATM, is that there is no easy way to map the memory in a protected manner simultaneously into multiple user spaces.

In contrast, with descriptor-based packet memory, it is possible in principle to support address mapping irrespective of DMA support. However, typically on descriptor-based PIO controllers, the existence of a small amount of on-board buffer memory makes it difficult to provide address mapping support, because that memory is a scarce resource that must be managed sparingly. For instance, the DECstation's Ethernet controller has only 128K bytes of buffer memory to be used for both send and receive buffers. Mapping pages of the buffer memory into user space would be costly, because the smallest units that can be individually mapped are 4 Kbyte pages. Reducing the number of available buffers this way could lead to delays due to dropped packets during periods of high load. On the other hand, conservatively managing the scarce buffer resource results in the kernel making an extra copy of the data from user space into the packet buffer.

With a trivial amount of controller hardware support, it is possible to solve the protection granularity problem, providing a larger number of individually protected buffers in controller memory. The basic idea is to populate only a fraction of each virtual page that refers to controller memory. As a concrete example, consider the following alternative design of the DEC Ethernet controller.

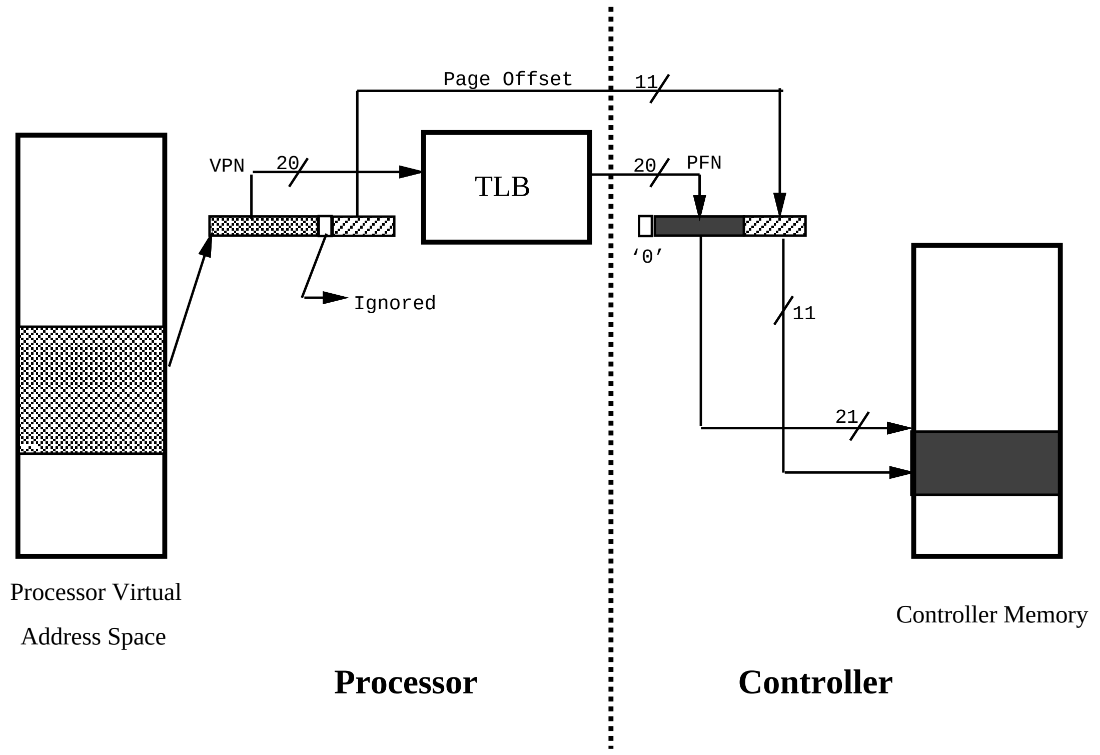


Figure 2.1: Address Mapping for Ethernet Buffers

Figure 2.1 shows the sketch of the design. Controller memory is organized as 2K byte buffers, each of which will hold an Ethernet packet; this would allow us to have 64 buffers in our 128K controller. To allow user processes to write directly to controller memory without sacrificing protection, the controller ignores the high order bit of the page-offset and concatenates it with the physical page number (PFN) field of each physical address presented by the TLB. This has the effect of causing each 2K physical page of controller memory to be doubly mapped into both the top half and the bottom half of a 4K process virtual page.

2.5 Chapter Summary

Our experience with host/controller interfaces leads us to believe that while simple FIFO-based controllers are ideally suited for small packets, larger packets would be better served with a more conventional descriptor-based packet buffer. Thus, it might be beneficial to support both forms on the same controller. To our knowledge, the only controller which has multiple host interfaces on board is the VMP-NAB [38].

Our experiments also suggest that, in the absence of memory system support, DMA may incur the cost of additional copies and/or cache flushing overheads. In such cases, it would be advantageous to use PIO with descriptor-based packet memory that the host CPU can either copy or map into user space. The decision to copy or map in will depend on whether the processor is *required* to touch every byte of the packet or not. For instance, if it is necessary to calculate a software checksum, then an integrated copy and checksum loop (as proposed in [17, 19]) would suggest that mapping is of limited benefit. However, on next-generation networks under certain conditions, there is evidence to suggest that a software checksum may be of limited use [75]. In this case, the processor does not need to mediate the transfer, and I/O-initiated data moves would be of benefit if the memory system provides adequate support for cache consistency. Fortunately, newer processor and memory architectures are providing these support mechanisms.

Finally, our experience with an existing ATM controller also suggests that it is appropriate to offload some amount of per-cell processing into the controller.

We expect the ideal controller for next-generation workstations and ATM-style local-area networks to have (1) hardware support for cell segmentation and reassembly, (2) a FIFO-based interface for single cell transfers and a coherent DMA-based interface for multi-cell transfers, (3) hardware support for checksums, and (4) a flexible interrupt structure that allows the host to be conditionally interrupted on cell arrival. At the present time, there are controllers that implement different subsets of these features. However, trends in high-performance controller design are promising and we expect controllers with low latencies to become widely available.

For the rest of the thesis, we assume the existence of fast controllers, high-speed networks, and fast processors. Given such high performance hardware, it is imperative that we design suitable network access models and distributed programming models (as described in Chapter 1) so that distributed services can benefit from advanced hardware. In the next chapter, we argue that current network access models may be ill-suited for building distributed systems, and we propose a different model. Subsequent chapters describe the implementation and the utility of the new network access model in various situations.

Chapter 3

A NEW NETWORK ACCESS MODEL

3.1 Introduction

Faster processors, high-speed networks, and next generation controllers promise high performance communication hardware in workstation clusters. However, in order to exploit the hardware performance, it is important that the software layers provide good performance as well. This chapter considers the design of the network access model, which serves as a low-level communication layer on which higher-level distributed programming models, such as RPC, are implemented.

In the past, people have predominantly used a network access model that views the network as a sequential stream of bytes, accessed with *sends* and *receives*. However, in this chapter we will argue that it is more beneficial to think of the network in terms of a memory abstraction. By this we mean that rather than using send and receive primitives to access remote data, programs *read* and *write* portions of memory located on a different machine.

The notion of remote memory access on a local-area network has been suggested before. Spector's work on the experimental Ethernet is perhaps the earliest example [61]. He suggested that by implementing a set of operations that could be performed efficiently over the network, we could lower communication overheads. The network access model we propose here is influenced by his work, but has several significant extensions to it. These are related to sharing, protection, independent time-slicing among nodes, and others. Spector's work, done on the early Xerox Alto workstations [65], left unaddressed important issues regarding protection and virtual memory, which are common concerns in current and next-generation workstations but were not supported by the Alto.

Previous work has also investigated support for network-wide coherent shared virtual memory either in hardware [25] or in software [41]. The network access model we propose here is quite different from these efforts. Our goal is to provide efficient primitives to access the network so that distributed programming models, such as shared virtual memory, can be implemented efficiently on top of it.

At a high level, all models of network access have the same power. For example, from a functional standpoint, it is immaterial whether a process chooses to affect the state of a remote process by sending it messages or by directly changing the contents of its memory. However, a network access model is more than just an abstractional convenience. From a pragmatic viewpoint, certain choices may be easier to implement than others. Further, certain classes of applications and of system organizations can be made more efficient by an appropriate choice of network access model. For example, we show that we can exploit the model presented in this chapter in a variety of ways for increased performance. First, performance of distributed programming models (e.g., RPC) that form the basis for distributed system communications can be improved by using the model. Second, distributed systems can be organized in a different and more efficient way without resorting to the traditional RPC-based approach. Further, the model allows efficient medium-grained parallel applications to be written for a workstation cluster. Chapters 4—7 explore these ideas in greater detail.

3.2 The Problem with Existing Network Access Models

Distributed programming models, which are layered over network access models, typically have the following requirements:

- Low data transfer overhead. Distributed programming models, including RPC, involve frequent data exchanges across machines. If the network access model does not support mechanisms for efficient data exchanges, overall performance could degrade.

- Low control transfer overhead. Typical distributed programming models require control transfers to occur at the sending and receiving machines. Control transfer can be a major component of the overall communication cost, and it is beneficial if the network access model permits this cost to be minimized.

- Low programming overhead. If the abstraction presented by the network access model is excessively complex to use, then the performance of distributed programming models could suffer.

Typical network access models, e.g., send/receive on a network byte-stream, do not always support these requirements very efficiently. For instance, send/receive primitives usually depend on messages exchanged between *communication end-points*, e.g., sockets or ports. Ports and sockets allow a degree of flexibility because they can be migrated among address spaces. However, they have the disadvantage that data is not sent directly to the final destination, which is either registers or a virtual address region in the recipient's address space. Instead, sockets and ports address data to intermediate buffers maintained by the communication system rather than the end application. Since the ultimate data destination is not specified in the message, some agent has to move the data to the eventual destination. This usually incurs processing overhead in the form of data copying (or page mapping) and input packet demultiplexing.

As a specific example, Unix-like operating systems provide the notion of a *socket* on which the data is sent and received sequentially. To receive data, the caller provides a buffer in which it expects the data from the sender. However, since the data from the sender is addressed to the socket and not to the buffer, the operating system stores incoming data in intermediate system buffers associated with the socket until the receiver identifies its buffer through a receive call. This has the effect of adding buffering and copying overhead, which can affect performance. Similarly, to send data, the user supplies a data buffer. However the data from this buffer is not directly sent on the network; instead, data is copied into an operating system buffer for eventual transmission.

It is possible to modify the send/receive interface to eliminate some of the problems with data copying, but at the cost of programming complexity. For instance, a send from a user could signify that the data buffer is freed from the user's address space, allowing the operating system to use the user buffer directly for network transmission. However, buffers "lost" by the user space will have to be replenished by the operating system, which requires another set of interfaces between the user and the system. Similarly, on receives, the operating system, rather than the user, can choose where incoming data is placed in user space. This is a compromise solution because the location may not be the most suitable for the user, i.e., the user might still incur the cost of moving data to a more convenient location.

Another typical feature of send/receive interfaces is that both sender and receiver have to be active participants. For example, a receive must be "posted" (either synchronously or asynchronously) and a send must complete if the data is to be examined by the recipient. Further, there is an asymmetric relation between the two communicating parties. For example, the receiver is not allowed to initiate a request; that must come from the sender. Many features of the send/receive network access model involve some form of handshake or synchronization between sender and receiver, and keep the sender and receiver at arm's length, even if they trust each other.

3.3 A Memory-Based Network Access Model

A network access model can be logically thought of at two levels. At one level, it can be thought of as the software interface provided to the user by the operating system, implemented using the facilities provided by the host architecture and the network controller. At another level, it can be viewed as the interface provided by the controller hardware itself. In practice, one would expect a hybrid scheme where the network access model is provided by a combination of controller hardware and system software. A natural way to divide the labor is to put the performance critical functions in the controller, and the rest in system software.

We describe below an alternative network access model, one that is *memory-based*. Unlike a send/receive model, a memory-based network access model directly supports reading and writing data to the final location, thereby minimizing data copying and demultiplexing costs. It provides protected and efficient user access to the network with support for virtual memory and multiprogramming. It incorporates a flexible scheme to minimize the cost of control transfer, shows good performance, and is a suitable interface on which to efficiently implement a variety of distributed programming models.

At an abstract level, this network access model consists of a set of remote memory *segments* and operations defined on them. Segments are contiguous pieces of user virtual memory; they are defined by user applications and controlled through *descriptors* maintained by the controller and privileged software. Applications exchange segment information through a higher level protocol. Once exchanged, descriptors can be named by the communicating parties, permitting direct *reads* and *writes* of data at specified offsets within the remote segments. Read and write operations are supported through special meta-instructions, described in detail below. Segments are protected from unauthorized access, because applications can selectively grant or revoke access rights to their exported segments.

It is most convenient to think of the access model as the instruction set of a communications co-processor. The model supports four non-privileged instructions: `WRITE` and its variant `WRITEV`, `READ`, and `CAS` (compare-and-swap).

The standard write instruction has the form: `WRITE` *rd*, *off*, *count*, *notify*. *Rd* specifies a descriptor register in the co-processor that identifies a remote memory segment. The descriptor includes the destination segment size, remote node address, and protection information. *Off* denotes the starting byte offset in the segment for the write. *Count* specifies the number of bytes to be written. The data for the write is taken from a set of message registers shared between the sending processor and co-processor. *Notify* indicates whether the remote destination is to be notified when the data reaches there.

On a write instruction, the co-processor verifies the rights of the sender. If the check is successful, it formats the data from the registers and sends it to the remote destination together with a descriptor identifier, offset, and count. If the network cannot be immediately accessed, the co-processor buffers the data in an internal FIFO, delivering an exception should the FIFO overflow. When a write successfully completes locally, the model only guarantees that the data has been accepted by the network, not that it has been delivered successfully to the destination.

The read instruction has the form: `READ` *rs*, *soff*, *count*, *rd*, *doff*, *notify*. *Rs* and *soff* specify the remote source segment and offset where the data to be read can be found. *Rd* and *doff* define a local destination segment and offset where the data is deposited. The `READ` request is non-blocking, i.e., the issuing process is allowed to proceed. When the data is returned from the remote processor, it is deposited in the reader's address space. No message registers need to be specified for a `READ`, which simplifies its operation. *Notify* indicates whether the reader should be notified when the read returns the data. In the absence of notification, the reader has no way of knowing the read returned data except by repeatedly checking the destination memory location.

The `WRITEV` instruction is a specialized variant of the standard write instruction that allows data to be scattered to remote memory. It has the form: `WRITEV` *rd*, *off*, *chunksz*, *xchunks*, *xstride*, *ychunks*, *ystride*, *notify*. The *rd*, *off*, and *notify* arguments are exactly as described for the `WRITE` instructions. The arguments *chunksz*, *xchunks*, *xstride*, *ychunks*, and *ystride* specify how the data is to be scattered at the destination. The total amount of data scattered with a single call is $chunksz \times xchunks \times ychunks$ bytes. As in the case of the `WRITE` instruction, the data is taken from a set of registers. On the destination node, remote memory starting at offset *off* is treated as though it is organized as a rectangular array of elements each of size *chunksz* arranged as shown below.

The `WRITEV` instruction is a special optimization expected to be used by scientific applications that manipulate 2- and 3-d arrays. Scientific applications that operate on 3-d

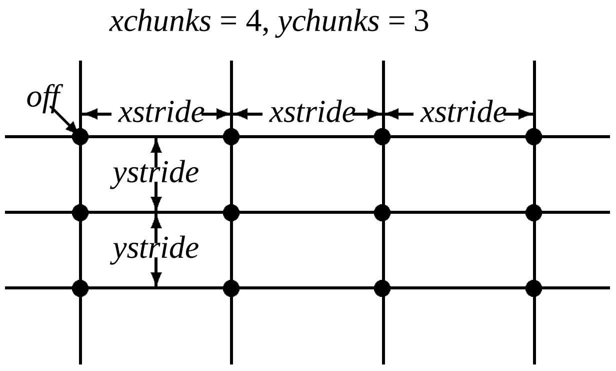


Figure 3.1: Three Dimensional Scatter Write

arrays frequently send data from an entire plane in the array to remote nodes. The 2-d grid shown above can be visualized as a particular plane of a 3-d array. Thus, using the `WRITEV` instruction, an application can write a set of physically non-contiguous locations in memory that correspond to elements of a plane in the 3-d array. 2-d arrays can be handled as a special case of 3-d arrays with *ychunks* = 1.

The `CAS` instruction is used to provide low level synchronization for communicating entities by way of an atomic compare-and-swap operation. It has the form: `CAS` *rd*, *doff*, *rs*, *doff*, *old-value*, *new-value*. *Rd* and *doff* specify the remote location (segment and offset) whose value is to be compared and swapped. *Old-value* specifies the value against which the contents of the remote location is compared. *New-value* denotes the new value that is to be atomically written in the remote location if the comparison succeeds. *Rs* and *soff* specify a local segment and offset that contain the result of the compare-and-swap, which is either success or failure. Our implementation of `CAS`, described in the next chapter, ensures that it is reliable under data loss in the network.

Beyond these simple instructions, the model explicitly supports additional mechanisms to accommodate the needs of applications. These mechanisms include (1) descriptor maintenance, (2) export and import of segments, (3) application-based pinning/unpinning of virtual memory pages, (4) segment write inhibit for synchronization, and (5) interrupt control. Support for these facilities is an important part of the model, and differentiates it from previous efforts.

3.4 Details of the Model

This section describes additional functionality of the model related to descriptor maintenance, export and import of segments, protected and shared access to network resources, application-controlled pinning of virtual memory, and interrupt control. For the purpose of explaining the details of the model, we will continue to view the model as the instruction set of a communication co-processor augmented by a few less frequently occurring operations provided by the operating system as system calls.

3.4.1 Descriptor Maintenance

In a workstation cluster it is important to ensure that applications that timeslice the cluster do not access segments that do not belong to them. In our model, accessing or modifying descriptors is a privileged operation; ordinary processes can only name descriptors as message sources or destinations. Descriptors are maintained by participating kernels which are trusted. This ensures that the necessary protection information is maintained securely. Descriptor maintenance is not on the critical path and thus it is appropriate to use reasonably sophisticated kernel software for this purpose. Certain predefined descriptor registers (and the associated segments) are reserved for the use of the kernel. Thus packets containing descriptor maintenance information can be exchanged between nodes using these reserved descriptors. This mechanism can be used, for example, by a name server to exchange information about segments that are exported and imported.

3.4.2 Exporting/Importing Segments

Segments are contiguous portions of the virtual address space that are named using descriptors. Before segments can be used by remote processes, they must be named and communicated between the participants. The model uses the simple notion of *exporting* and *importing* segments by name. No semantics is imposed on the names, which can be chosen in any way that the exporter sees fit. For instance, long 128 bit integer strings can be used to guard against the possibility of an unauthorized agent guessing a name. Additional protection is ensured by specifying *protection groups*. A protection group names a collection of users that trust each other.

To export a segment, a process executes the following: `EXPORT` *segment-name*, *virtual-address*, *size*, *access-type*, *protection-group*. *Virtual-address* and *size* specify the start and bounds of the segment. *Access-type* specifies the type of access (read or write) that is permitted and whether the exporter is to be notified on data arrival. *Protection-group* identifies a group of users who are allowed access to the segment. The `EXPORT` operation returns a descriptor number to the caller which can be used later to revoke the export by calling `UNEXPORT` *segment-id*. To import a segment a process executes: `IMPORT` *segment-name*. The import operation returns the handle of a descriptor that can be used for subsequent write and read operations on the exported segment.

Import and export, being less frequent operations that require the facilities of the operating system, are expected to be supported by system calls. The kernel on the exporting processor allocates a descriptor, which is initialized with the size and protection information. When an import request is received by an exporting kernel it ensures that import rights are not granted outside the protection group. A reference to the exported segment is returned to the importing kernel, which in turn allocates a descriptor and returns a reference to the caller of `IMPORT`.

The model makes no assumptions on how the importer names, locates, or contacts the exporter. Nevertheless, it provides hooks by way of privileged, predefined descriptors (and segments) to build a self-contained name server that uses no other facility other than what is provided by the model. In fact, this is precisely how our name server is implemented. However, the model does not preclude the use of an existing facility to name segments, e.g., by using a distributed file name hierarchy.

3.4.3 Virtual Memory

On multiprogrammed workstations, it is necessary to provide for the case when parts of a segment are not memory resident. The alternative of pinning entire segments is infeasible for realistic segment sizes. On the other hand, faulting in pages of the segment at message reception time could lead to unacceptable delays. The model allows the application control over its exported segments by making it possible to selectively (and dynamically) pin portions of each exported segment. Programs can use their knowledge of locality to exploit this facility. In our scheme, since each application pins a small amount of memory, the operating system's pool of pinned network buffers can be reduced. That is, physical memory usage is comparable to a traditional organization where the operating system pins a common buffer pool for all applications.

Pinning and unpinning portions of a segment are done by issuing the calls: `PIN` *segment-*

id, *addr*, *size* and `UNPIN` *segment-id*, *addr*, *size*. These operations are relatively complex because they involve the virtual memory system. However, they do not occur frequently enough to be a performance bottleneck. Consequently, typical implementations of these operations (including the one described in the next chapter) will be within the operating system and not in the co-processor.

On receiving data to be written, the co-processor consults address translation tables to translate the virtual address in the message to a physical address. If there is no translation or if the addressed portion of the segment is not resident, the data in the packet is written to a circular queue. Higher level software in the kernel maintains the buffer queue and multiplexes data to the correct address. In certain environments, the simpler alternative of dropping the packet altogether may be feasible; for example, if there are higher level transport protocols that assume that the network is unreliable. On a data read request where there is no address translation or if the data is not resident, the request is ignored.

3.4.4 Memory Volatility

The notion of remote reads and writes effectively creates regions of memory that are volatile. By this we mean that the contents of a memory location in an address space can change without the address space issuing a store instruction to that location. Usually this is not a problem if the programming language and compiler support the notion of *volatile* data, e.g., as found in C. However, there are cases where memory volatility could be inconvenient or error-prone to use. To address this problem, the model allows the application to control the window of volatility. An application can turn write protections on and off to segments using lightweight operations implemented by the co-processor itself. These operations are invoked with the following co-processor instructions: `MPROT` *rd*, and `MUNPROT` *rd* respectively to turn off and turn on write permission on the segment named by *rd*. A write request arriving for a protected segment is treated exactly as if there were no address translation for that segment, i.e., the kernel buffers it in a circular queue.

3.4.5 Control and Data Transfer

Control transfer and data transfer are separated in our model. For instance, when data arrives at the destination it is written to memory but the destination is not automatically notified. Recall that the `READ` and `WRITE` instructions have a bit called *notify* that is used to provide control over notification. In addition, each segment descriptor contains a notification control flag that can be set by the host in one of three states: (1) always notify, which causes the destination to be notified whenever a packet destined for that descriptor arrives, (2) never notify, which causes the destination to be never notified, and (3) conditionally notify, which causes the destination to be notified only if the *notify* bit is set on the remote request instruction.

3.5 Chapter Summary

In this chapter, we have argued that traditional network access models, such as send/receive, have some limitations that can be overcome by using a new memory-based model. To summarize, the new network access model that we propose has several key advantages.

First, the notion of *local memory* is a very natural and well understood one. The model and the primitives we propose extend this notion to *remote memory* as well. As we will show in subsequent chapters, there is a simple implementation of the ideas presented in this chapter, which ensures that programs can access remote memory almost as easily as local memory.

Second, the model of remote access is *passive*. That is, a remote operation, such as a read or a write, does not involve activity on the part of the target address space. For instance, a write operation can complete without involving the destination address space. However, in certain cases, it is important to start the execution of a thread or a particular procedure at the destination. The model provides this as a separate option. Thus, the transfer of data and the transfer of control are separate. This has an important implication: since control transfer and data transfer are separated, both *function shipping* (e.g., RPC) and *data shipping* (e.g., bulk data transport) mechanisms can be independently optimized. This has many ramifications for the structure of distributed systems, which we expand on in Chapter 7. Previous memory-based approaches, e.g, Spector's work, have not pursued these ideas or studied their impact on distributed system structure.

Third, the model ensures that each request specifies the ultimate data destination. This allows us to minimize the number of data copies and reduces the overhead of segmentation and reassembly in modern cell-based ATM networks.

Finally, the model allows users very efficient, protected, and shared access to the network. In addition to memory instructions, the model provides support functions that are necessary on modern timesharing workstations. Together, they ensure that the model can be used as an efficient base for implementing a variety of distributed programming models.

The network access model described in this chapter has been implemented on an ATM network. We describe this implementation in the next chapter. While we believe the model can be implemented on any network, it has some particularly nice properties in the context of current and next-generation ATM networks.

First, ATM networks offer us the ability to efficiently transfer small amounts of data in the form of cells. This meshes well with the notion of loads and stores supported by our network access model. Second, switch-based local-area ATM networks are getting extremely reliable. For instance, there are ATM networks currently in existence that guarantee against packet loss once a packet is injected into the network [4]. This implies, for example, that a set of cooperating applications distributed on a local-area network can use the network access model as though they were cooperating processes sharing memory on a single processor. Finally, it is clear that as ATM networks speed up into the gigabit range, the cost of transferring data is going to be significantly lowered. However, the cost of control transfer, which involves context switching overheads, does not appear to scale with processor speeds [3]. Thus, a network access model such as ours, which separates the notion of control transfer from data transfer, is a key advantage in next-generation workstation cluster environments.

Chapter 4

AN IMPLEMENTATION OF THE MODEL, AND ITS PERFORMANCE

We have implemented the memory-based network access model using a software emulation layer on top of an existing ATM controller attached to off-the-shelf workstations. The implementation is optimized around the expectation that, in general, (1) many operations are performed on a segment once descriptors are exchanged, and (2) most operations transfer a relatively small amount of data, e.g., on the order of tens of bytes. In this section we briefly summarize the characteristics of the controller and the host architecture before describing our prototype implementation.

4.1 The Testbed

We use the FORE TCA-100 network controllers described in Chapter 2 to connect our DECstation 5000s to a 140 Mbits/sec ATM network. Recall that the controller is located on the 25 MHz TURBOChannel I/O bus and does not have DMA capabilities. Instead, it implements two FIFO queues, one for transmitting ATM cells to the network and the other to buffer received cells. The processor transmits and receives cells by performing word-aligned read and write instructions to memory-mapped I/O addresses. Processor accesses to the FIFOs complete without any wait states.

ATM cells are 53 bytes long and are divided into a five byte ATM header and a 48 byte payload. The five byte header includes an eight bit header CRC and the payload includes a separate ten bit payload CRC. Both CRCs can be optionally computed by the controller hardware; our implementation enables only the header CRC.

The processor transmits a 53 byte cell by performing 14 word writes to the transmit FIFO. The three ($14 \times 4 - 53$) spare bytes are used to pass various cell processing options to the controller, e.g., to enable or disable checksums. When the 14th word write is performed, the cell is transmitted on the wire. Processing of multiple cells is pipelined. That is, while the processor is writing data for a cell to the FIFO, the controller can potentially be transmitting data from a previous cell. The controller is capable of transmitting smaller sized ATM cells but we do not use this feature because it is non-standard.

The processor receives each 53 byte cell by performing 14 word reads over the TURBOChannel. The three spare bytes include framing information and the CRC syndrome bits for the header and payload. These CRC syndrome bits can be examined by the host to correct single-bit errors in the header and the payload. In addition, the processor can detect a CRC or framing error with a single comparison instruction.

The controller design makes it difficult to use the payload CRC efficiently on receives. The hardware design enforces a FIFO access discipline not only on the individual cells but also on words within a cell. Thus, since the ATM payload checksum is at the end of the cell, all prior words have to be read from the FIFO and buffered before the contents of the cell can be used safely. In the common case of no errors, this buffering cost represents unnecessary overhead. Because of this, and because fiber optic links have very low error rates (1 bit in 10^{12}), we have chosen not to employ the payload checksum in our prototype implementation. A simple change in the hardware design that allows random access to the individual words within a cell would solve this problem.

The transmit FIFO is 512 words deep and can store 36 cells. If the transmit FIFO fills up, hardware flow control in the controller will cause the processor to stall. The receive FIFO is deep enough to receive 292 cells. Cells that overflow the receive FIFO are dropped.

ATM cells have a fixed small size to minimize variance in queueing delays in a switched network. Since cell size is quite small, ATM networks define adaptation layers, called ATM Adaptation Layers (or AAL) that specify how cells are fragmented and reassembled from larger packets. There are two AALs that are being proposed as standards — AAL 3/4 and AAL5. AALs have two sublayers called the segmentation and reassembly (SAR) layer, and the convergence (CS) layer. Typically, the function of the SAR layer is to handle the reassembly of cells, while the CS layer checks the final result [32].

There is hardware support for some AAL 3/4 functions, such as CRC generation, in the FORE controller. However, there is no support for the SAR or CS layers in hardware. The device driver is capable of handling reassembly and fragmentation in software. We have measured a latency of 11 μsec per cell for the SAR/CS function in software. Our implementation of remote memory uses neither the SAR layer nor the CS layer and is not fully compatible with the device driver's use of AAL 3/4. That is, although the kernel contains code for both the device driver and our implementation of the network access model, they cannot be used simultaneously. There is a system call that allows users to switch between one and the other. However, by giving up one word of payload from the cell and modifying the device driver, we can operate simultaneously. This change adds only three instructions on the send and receive side and imposes no observable latency reduction. Throughput will be decreased by 10% because there is a 10% decrease in payload.

The DECstations contain 25 MHz MIPS R3000 processors rated at 18.5 SPECmarks. Many exceptions, including all I/O interrupts, are vectored to a single handler, which must then identify both the type of the exception and the originating device for interrupts. This requires reading the system's control and status registers as well as accessing the on-chip control co-processor's registers. Thus, servicing an interface interrupt requires instructions in addition to those required to handle the device specific operations.

4.2 The Prototype Implementation

In the prototype, performance critical remote operations have been implemented by emulating unused opcodes in the MIPS instruction set. That is, the application performs a remote operation by issuing an unprivileged and unused MIPS machine instruction. The instruction is emulated in the kernel using assembly language instructions. Less frequent operations of the model such as import and export of segments are handled by the operating system through standard system calls. The prototype is hosted in an otherwise unmodified Ultrix operating system.

To the user, network access is through user-level instructions; nevertheless the network is protected since only kernel code directly manipulates the network. We assume that all the kernels that are connected to the local ATM network are trusted. These trusted kernels also ensure that exports and imports of segments are done only between consenting principals. There is a name server, trusted by the kernel, that cooperates with it in managing segments.

The prototype differs from the model in two minor ways. These differences made the implementation simpler without sacrificing functionality and without artificially distorting the performance of the resulting system.

The first difference is related to the way in which control is transferred on cell arrival. The model assumes that there is a co-processor that implements the instructions and that the processor is notified only if the notify bit is set in the cell. In our prototype, since we use software on the host processor to emulate the co-processor, cell arrival causes activity on the *processor*. However, the destination *process* will not be notified unless the notify bit is set in the cell. Note however that this does not mean that the processor is interrupted on every cell arrival because the TCA-100 hardware interface allows the processor to handle multiple cells with a single interrupt.

Second, the current prototype does not implement the scatter write instruction `WRITEV` described in the previous chapter.

The next few subsections supply implementation details of the prototype.

4.2.1 Exporting a Segment

A segment is exported by an application by making the `rr_export` system call. The declaration and usage of the call is shown in the C code fragment below.

```
char *rr_export(char *name, int sz, int prot, int *segment,
                char *addr);
char *addr;
int  segment;

addr = rr_export("ServiceFubar", 8192, 0660, &segment, 0);
if (addr == (char *) -1) {
        perror("export failed");
        exit(1);
}
```

The argument `name` specifies the name of the exported segment that an importer will subsequently use to refer to this segment. By specifying a null value for `name`, it is possible to export anonymous segments that can never be imported by anybody. Anonymous segments are typically used as destinations of read and compare-and-swap requests. The argument `prot` specifies access restrictions that apply before programs can import the segment. We use the standard Unix convention for specifying file access protection. Thus, 0660 in the example above means that importing applications executing within the same group and owner may import the segment for reading and writing. The exporter specifies the size of the segment in bytes with the `sz` argument. The return value specifies the virtual address of the newly created segment in the caller's address space. The variable `segment` is the segment handle that can be used by the exporter to perform control operations on the segment such as virtual memory operations and subsequent unmapping operations as described below. The last argument to `rr_export` specifies a location at which the exporter would like the segment to appear in its virtual address space. Specifying zero, as in the example above, leaves the choice up to the system. A non-zero value, if specified, must specify a valid page-aligned virtual address. An exported segment can be deleted with a `rr_cleanexport (int segment)` call.

Another example of the usage of the call is shown below, this time with a non-zero address. Here, a 16 Kbyte region in the heap, aligned on a 4 Kbyte page boundary, is allocated and the first 8K bytes are exported.

```
char *new,*addr;
int  segment;

addr = malloc(4 * 4096 + 4096);
addr = (addr + 4096) & ˜(4096-1);
new = rr_export("ServiceFubar", 8192, 0660, &segment,addr);
if (addr != new) {
        perror("export failed");
        exit(1);
}
```

The Segment Descriptor Table

A successful `rr_export` operation allocates an entry, called the segment descriptor (SD), in a kernel table named the Segment Descriptor Table (SDT). The SDT contains one segment descriptor per exported segment on the machine (up to a maximum of 2048). In the current implementation, to save memory resources, the SDT is only large enough to hold eight SDs.

An SD within an SDT on a particular machine is identified by its index into the table, called the SDI. The SD contains the process identifier of the exporter, protection bits for the segment, the size of the segment, and a table of page-mappings. Each SD also contains a 12 bit generation number that is incremented each time a new export reuses the SD.

The generation number in the segment descriptor is used to detect incoming remote operations that refer to old exported segments that have since been reused. That is, each incoming request contains a generation number that is compared against the generation number in the segment descriptor. The request is discarded if the generation numbers differ. Since there is only a finite number of bits for the generation number, it will wrap around eventually. Special care has to be taken to ensure that requests using old generation numbers do not cause problems. The mechanism for this is straightforward and is described in the section about the name server (Section 4.2.6).

The SD does not contain the user-specified name of the segment. Instead, the user-specified name, the SDI, and the protection information about the segment are registered with a name server. Certain ATM-specific header information is also registered so that an importing kernel can set up suitable headers for transmission over the ATM network.

Each SD contains a table of page mappings — one per page of the exported segment. Each mapping contains either a kernel virtual address to a pinned page frame or a null entry indicating that the frame is not pinned. Notice that there may be pages in the segment with resident, but unpinned frames. The mappings corresponding to these pages are also nulled. In the prototype, the maximum size of an exported segment is 1 Mbyte, which requires 256 map entries corresponding to the 256, 4 Kbyte page frames. The kernel virtual address in each mapping is consulted by the kernel when an incoming packet is processed to perform the requested remote load or store.

4.2.2 Importing a Segment

Segments registered with the name server can be imported by the kernel on behalf of users. Typically, segments are imported as shown below.

```
#include <file.h>
char *rr_import(char *name, int prot);
int  segment;

segment = rr_import("ServiceFubar", O_RDWR);
if (segment == -1) {
        perror("import failed");
        exit(1);
}
```

Notice that there is no virtual memory created in the importer's address space. Instead, a segment handle is returned. Virtual addresses are constructed and passed to the read and write routines as <segment, offset> pairs. An imported segment can be subsequently deleted with an `rr_cleanimport(int segment)` call.

Before an `rr_import` call can succeed, it is necessary for the importing kernel to contact the name server on behalf of the user. The caller is blocked until the required information is returned by the name server, or there is an error. The name server returns an error if the named segment does not exist or if the importing user does not have the requested privileges on the segment.

A successful `rr_import` call allocates an entry in a kernel table called the Registry. Each Registry entry (RE) contains information about the local importing process as well as network header information, generation number, and the SDI of the remote segment. The header information, the SDI, and the generation are sufficient to address a specific segment on a particular node.

Note that the size and protection information of the remote segment are not maintained in the RE. The reason for this is that the local kernel does not check size and protection on outbound transmissions. Even if it were to enforce size and protection constraints, the receiving kernel would have to do a similar check again to verify if the generation was stale, whether the offset referred to a currently pinned region of the segment, and whether the exporter had changed access permissions (to be described in Section 4.2.4) on the segment.

4.2.3 Remote Operations

After a segment number has been obtained by a `rr_import` system call, remote operations can be performed from user-level by issuing an unused MIPS instructions as described below.

Remote Write

There are two types of remote writes — `rwi` and `rwblk`. The instruction `rwi` always writes a predetermined amount of data (viz., ten words) to the remote segment. The second form, `rwblk`, takes a variable amount of data.

The `rwi` instruction has the form `rwi $r4, $r5`. We use the convention that $rN denotes register number N on the MIPS processor. Register $r4 contains the segment handle returned by the `rr_export` call. Register $r5 contains the offset and notify bit for the request. The notify bit is the MSB of the 32-bit register. If set, the remote write can potentially cause a control transfer on the remote end if the exporter is ready for notifications. If either the bit is not set or the exporter is not ready for notifications, only data transfer takes place. The `rwi` instructions looks for the ten data words in registers $r6–$r15.

To perform block writes of variable length, the instruction `rwblk`, which has the form `rwblk $r4, $r5, $r6, $r7` is used. Registers $r4 and $r5 contain the same information as before. Register $r6 contains the byte count and register $r7 points to a block of data in (local) memory that is used in sequence until the byte count is satisfied.

Remote Read

Like the remote write, there are two types of remote read instructions — `rri` and `rrblk`. The first, `rri`, reads a fixed amount of data (ten words) from a remote memory segment. It has the following format `rri $r4, $r5, $r6, $r7`. Register $r4 contains the segment number of the remote segment, $r5 contains the segment number of an exported local segment. Typically this is an anonymous segment. Register $r6 contains the offset in the remote segment to load the data from and register $r7 contains the offset within the local segment to store the data. Register $r7 contains the notification bit exactly as in the case of the remote write operation previously discussed.

To specify a block data transfer, we use the instruction `rrblk` as follows: `rrblk $r4, $r5, $r6, $r7, $r8`. Registers $r4 – $r7 contain the same information as before and $r8 contains the byte count.

A remote read is treated as a read request to the remote node followed by a remote write operation initiated by the remote node in turn. This makes it easy to implement remote read operations without any additional machinery beyond that required for remote writes.

For user convenience, various C-callable assembly routines have been provided that take pointers to buffers and word counts. There are also routines that optionally set the *notify* bit on reads and writes. Thus, normally programmers do not deal with loading data into the correct registers and setting the appropriate bit for notification on the remote end.

Remote Compare and Swap

The usual semantics of CAS (compare-and-swap) are as follows. Let A be a memory address, O and N be registers. Then performing a `CAS A,O,N` has the effect that if the contents of A matches the contents of O, then it is swapped with the contents of N, and CAS returns the boolean `TRUE`. The comparison and the swap with the new value is atomic. If the contents of A does not match the contents of O, then CAS returns `FALSE`.

It is important to note the guarantees offered by a CAS operation. For example, consider a multiprocessor that implements CAS. If a particular processor performs a CAS that returns the value `TRUE`, then that processor cannot really determine the *current* contents of the CAS location. This is so because another CAS could have overwritten the new value. In effect, all that the CAS guarantees is that at the time of the swap, the value in the location matched what the processor expected it to be.

Instead of CAS, we implement a primitive (which we shall call CAS-II) that has the following semantics: if the contents of A matches that of O, then swap with the contents of N as before, otherwise do nothing. As before, the comparison and swap are indivisible. But, whereas standard CAS returns true or false, we return the value contained in A at time of the comparison and the identity of the entity that did the last successful swap.

Using CAS-II, a processor can have the same guarantees that CAS provides. For example, a processor P1 can get the semantics of CAS using CAS-II in the following way.

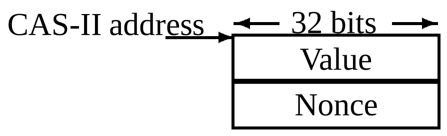
```
CAS (A, O, N)

X = record
   int processor;
   int old-value
end X;

X = CALL CAS-II with <A, O, N>
if X.processor is P1 and X.old-value matches the contents of O,
   then return true
else
   return false
end CAS
```

Given a reliable network, CAS-II as defined above can be implemented directly. However, in the presence of network cell loss, we need to modify it slightly in the following fashion.

Implementation of CAS-II

A process wanting to do a CAS-II issues the `rrcsi` instruction. The `rrcsi` is two-phase, i.e., a processor issues a CAS-II request on a remote memory location and can later check a local memory location for success or failure. This allows useful computation to be overlapped with the CAS-II operation.

Each CAS-II address location refers to a 64-bit region that is organized as two 32-bit words as shown below.



The first word holds the value that is used in the CAS operation. The second word contains bookkeeping information as described below.

The exact format of a CAS-II instruction is: `rrcsi $r4, $r5, $r6, $r7, $r8, $r9, $r10`. The register pair <$r4,$r6> specifies the segment and offset of the address used for the compare-and-swap. The register pair <$r5,$r7> specify the segment (typically an anonymous segment as in the case of remote reads) and offset to be used to write the result of the CAS operation. Registers $r8 and $r9 contain the old and new values to be compared and swapped. Register $r10 is special, it contains a 32-bit nonce that the sender can use to match responses. (The nonce is usually a concatenation of a process identifier, parts of the host IP address, and a sequence number.)

The kernel emulation code on the requesting node forwards a single cell containing the CAS-II request parameters to the destination node. The cell contains information about the local address (specified in <$r5,$r7>) where the returned result should be written. The requesting user process is allowed to continue without blocking.

If the request cell is lost on the outbound transmission, the sender will eventually timeout and retry again with the same nonce as before. Note that losing an outbound request cell poses no problem to the semantics of CAS because it is as if no CAS request was made. Also note that it is not the emulation code in the kernel that is doing the retry; that is the responsibility of the user (or the user runtime system). Ultimately, a CAS-II request will reach the destination node that has exported the segment.

When the request cell reaches the destination node, the kernel emulation code on the receive side tries to perform a compare-and-swap. This operation is made indivisible by making sure that all CAS-II operations on a segment always go through the kernel. If the kernel is successful in the compare-and-swap operation, the incoming nonce is written to the second word pointed to by the CAS-II address. If the compare-and-swap fails, then the kernel does not modify either of the words. In any case, the kernel sends a reply cell, containing the latest nonce and the old value of the location, back to the requester.

If there is no cell loss, the result of the CAS-II operation will reach the requester's kernel where it will be written at the local address specified by <$r5,$r7> in the original `rrcsi` instruction. The result consists of two words of data — the old contents of the remote location, and the latest nonce. If the reply cell is lost, the requesting process (or the runtime) will retry the request with the same nonce until a reply is received.

There are basically two possible outcomes of the reply. If the nonce in the reply matches the nonce in the request, then the caller knows the compare-and-swap succeeded. The value returned will match either the value in $r8 (the old value) indicating that no reply cell was ever lost, or the value in $r9 (the new value) indicating at least one reply cell was lost. If the nonce does not match, then the compare-and-swap failed.

We mentioned that the CAS-II address refers to a <segment,offset> pair. Usually this is a remote segment and offset, except to the exporter of this segment, for whom the CAS-II address is actually a local memory address. However, the exporter cannot do a simple compare-and-swap operation on this memory location because the MIPS processor has no compare-and-swap instruction. In the absence of such an instruction, the exporter would have to use multiple instructions to simulate a compare-and-swap, during any one of which a remote CAS-II request could cause the processor to be preempted from the exporter. In order to maintain indivisibility in the face of network interrupts we force even local CAS-II requests to take a kernel trap to emulate code for the local case. A trap on the MIPS architecture automatically turns off interrupts until explicitly reenabled or until the return from trap instruction is executed.

Notice that in our implementation, we are relying on the participating user-level processes (or their runtime systems) to supply nonces. The kernel emulation code does not enforce or care about the integrity of these values. This is because we assume that entities that are using CAS-II on a particular location are cooperating to solve a problem. That is, they are not malicious and will follow an agreed upon protocol. Failure to do so does not result in any damage to the system; the applications simply hurt themselves by not being able to perform compare-and-swap. Note also that unauthorized access to a CAS-II value location *is* prevented by the kernel through the normal import/export security mechanism and segment access checking.

Apart from compare-and-swap, which provides synchronization support, our implementation offers an additional atomicity guarantee that can be used for synchronization. The implementation guarantees that a single-word local access (read/write) is atomic with respect to a remote access (read/write) that involves that word.

4.2.4 Local Virtual Memory Operations

Since segments are large and are impractical to pin down indefinitely, we allow the user to pin a small number of pages (currently up to four) per segment at any given time. By being able to selectively pin a region of memory, users have some control in ensuring that incoming remote requests are correctly delivered without any heavyweight kernel operations. Pinning and unpinning of memory is done by a common system call as follows:

```
char *rr_remap(int segment, char *unpin, int unl, char *pin,
        int nl);
```

The first argument refers to the segment returned by a prior `rr_export` call. The arguments `unpin` and `pin` point to virtual addresses within the exported segment. Arguments `unl` and `nl` each specifies a length in bytes. A contiguous portion of the segment starting at `unpin` and `unl` bytes long is unpinned first. Next, a portion of the segment starting at `pin` and `nl` bytes long is pinned. Pinning and unpinning are done on a page basis even though the addresses and counts are based on bytes.

The notion of remote operations introduces memory areas that are volatile. That is, memory local to a processor can be affected without the processor performing a local store. Since this might be undesirable in certain applications, the user can set the protection on an entire segment to read only or read-write. Protection is set by invoking the following system call: `char *rr_setprot(int segment, int prot)`. The second argument `prot` is specified using the standard Unix convention for specifying file access protections, exactly as described in Section 4.2.1.

4.2.5 Control and Data Transfer

As described in the previous chapter, our model explicitly separates the notions of control transfer and data transfer. Under the normal pattern of behavior, remote requests do not require user-level processes to take any action. However, there are times when it is useful to activate some user level activity in the address space. For example, this could be used to initiate an RPC request on a server.

Associated with each segment are two file descriptors that the user can access using the `rr_getfd` system call as shown below.

```
#include <rrf.h>
int rr_getfd(int segment, int type);
int datafd, controlfd;

control_fd = rr_getfd(segment, RR_CONTROL);
data_fd = rr_getfd(segment, RR_DATA);
```

The file descriptor `control_fd` becomes ready for reading when a remote operation causes a notification as determined by the settings of notification flags on the segment and the notification bit in the incoming request. For each such remote operation, a record indicating the type of the request and the location in the user's address space at which the request happened is made available on the file descriptor.

The file descriptor `data_fd` is provided to handle exceptional cases when a remote operation arrives for a portion of the segment that is not currently pinned by the application. Once again, a record describing the remote request, the location in the user's virtual address space and the actual data are made available on the file descriptor. The application can incorporate such data into its computation in any way appropriate.

We use file descriptors for notifications because they allow Unix applications to deal conveniently with asynchrony. Applications use the standard Unix "`select`", "`read`", "`fcntl`", and "`signal`" calls to receive notifications without having to poll for data arrival. In an operating system such as Mach that supports asynchrony within a process with multiple threads, control transfer can instead be implemented with upcalls and a shared page for transferring control information.

The file descriptor returned by `rr_getfd` can be used like any other Unix file descriptor; in particular it can be duplicated via the "`dup`" system call and inherited across "`fork`" system calls. However, the SDT entry corresponding to the file descriptor is not duplicated or inherited across `fork`s and does not survive the death of the process owning the SDT.

Thus, if process B inherits a descriptor from process A, which exported the segment, and A exits, then subsequent incoming remote requests to the segment will be ignored. However, process B can use the inherited file descriptors to continue processing data that was transferred while A was still alive.

An exporter can set the notify flag on the segment in one of three ways to regulate how control transfer (if any) is effected when a remote request arrives for the segment. This is done using the following system call: `rr_setintr(int segment, int notifytype)`. The value of `notifytype` can be one of the manifest constants RR_NOINTR, RR_INTR, or RR_CINTR. These respectively indicate that a notification is (a) never generated, (b) always generated, and (c) conditionally generated when a remote operation is performed on the particular segment.

4.2.6 The Name Server

The purpose of the name server is to provide a repository of segment names so that importers and exporters can communicate. The name server provides mechanisms to add exported segment names, to lookup names, and to delete names.

The name server maintains a list of named segments, their protection attributes, and the ATM network information required to get to the exporting host. The name server is trusted and privileged and is implemented as a collection of user-level processes distributed across the network.

Communication between the various processes that implement the distributed name service is implemented using the remote memory model itself. Certain well known segment identifiers have been reserved on each machine to allow the name service to bootstrap itself. We defer a complete discussion of the name server until Chapter 7.

As described earlier, generation numbers are used to detect stale remote operation requests. Segment descriptors are allocated in a round-robin fashion; thus, with a 12 bit generation field and 2048 entries in the segment table, generation wrap-around for a particular segment will occur after 2048×4096 exports. Assuming that segments are exported once a second, this wrap-around takes about 97 days. At or before each wrap-around, a kernel that is potentially using an old, invalid, segment descriptor should refresh its descriptors from the name service. Given the slow wrap-around rate, the name service information can be propagated fairly slowly without any major impact on performance or scalability.

4.3 Performance of the Prototype

Table 4.1 summarizes the instruction counts for remote read `rri`, write `rwi`, and compare-and-swap `rrcsi` instructions. As indicated in the first row, the cost of crossing protection boundaries is only 25 instructions. While the instruction counts for each operation are quite small, we still pay a price for the software implementation. One cost is the penalty incurred by the interrupt dispatcher in decoding the exception, which accounts for 28%–33% of the total instructions. Another cost is not evident from the table: nearly 50% of the instructions perform memory loads or stores, which tend to slow down execution. Despite these factors, a remote write operation containing one ATM cell (40 data bytes) takes only 30 μs. A remote read takes longer than a write—45 μs—since one cell must be sent in each direction. A remote compare and swap is slightly faster (38 μs) because there are fewer memory accesses on the sending and receiving sides.

If the cell size were increased, the *incremental* cost per word would be 3 or 4 instructions. In addition, Table 4.1 represents the worst case, because every cell arrival need not cause

Table 4.1: Instruction Counts for Remote Memory Operations

Number of Instructions for a 40-byte Remote WRITE Operation	
Operation	**Inst. Count**
Protection Crossing	25
Access Check & Writing Data to Device	40
MIPS Interrupt Dispatcher (on Destination)	73
Device Interrupt Handler	37
Reading Data and Writing Destination Addr	85
Total	260

Number of Instructions for a 40-byte Remote READ Operation	
Operation	**Inst. Count**
Protection Crossing	25
Access Check & Writing Read Request to Device	65
MIPS Interrupt Dispatcher (on Destination)	73
Device Interrupt Handler	37
Reading Request, Accessing Memory & Writing Data to Device	47
MIPS Interrupt Dispatcher on Source	73
Device Interrupt Handler	37
Reading Data and Writing Destination	85
Total	442

Number of Instructions for a Remote CAS Operation	
Operation	**Inst. Count**
Protection Crossing	25
Access Check & Writing Read Request to Device	70
MIPS Interrupt Dispatcher (on Destination)	73
Device Interrupt Handler	37
Performing Request, & Writing Result to Device	97
MIPS Interrupt Dispatcher on Source	73
Device Interrupt Handler	37
Reading Data and Writing Destination	63
Total	475

Table 4.2: Performance Summary of Remote Memory Operations

	Read	**Write**	**CAS**
Latency (μs)	45	30	38
Throughput (Mb/s)	35.4		
Notification Overhead (μs)	260		

an interrupt. If cells arrive back-to-back, only the first arrival would interrupt; the interrupt handler would then drain the input queue of all packets without further interrupts. Further, when streaming multiple cells, sending and receiving code can be pipelined.

We summarize the performance of our implementation in Table 4.2. The latency represents the elapsed time for performing single-cell accesses. Throughput is measured using the block write primitive on 4K byte blocks (the block read yields essentially identical performance). The notification cost is the overhead, in addition to the read/write request, that is incurred when the notification bit is set in an operation. The measurements shown are between two hosts connected directly without a switch; we expect next-generation switches to introduce only small additional latency. (The cost of notification is relatively high because of our reliance on the Ultrix signal handling mechanism.)

It is important to note that although the FORE ATM network has a bandwidth of 140 Mb/s, the best achievable memory-to-memory throughput on the DECstations with the FORE controller is considerably less than this. Our implementation achieves 70% of what would be possible if we directly used the FORE controller to transfer data to the memory of a remote machine.

It is instructive to compare these measurements with the raw performance of the underlying FORE network controller. To obtain this best-case performance, we constructed a simplified environment, ignoring issues of protection, sharing, and demultiplexing of incoming packets. The device was mapped into an address space, which accessed the network directly. We removed interrupt handling costs as well, because the address space polls the network constantly. Running two hosts in this configuration, we measured the time to do a write operation at about 11 μs. Thus, our software implementation has added about 19 μs to the write operation in order to multiplex multiple users and provide protected accesses. For read operations the increased cost is greater (about 23 μs), because the interrupt handling code is invoked twice, once on each end of the transfer.

One might ask whether a design optimized for cell-size writes suffers on larger block transfers, relative to a specialized block transfer primitive. To answer this question, we measured our block transfer instructions (`rwblk` and `rrblk`) that transfer data from an arbitrary-sized user buffer rather than a fixed set of registers. We measured the throughput achieved for a 4K remote block transfer to be 35.4 Mbits/sec, compared to 34.7 Mbits/sec using single-cell remote writes. Thus, a single-cell remote operation interface loses little in the way of throughput relative to block transfer primitives.

The reason for the behavior of block transfers is the following. On the sending side, only a single emulation trap is made for block transfers, compared to the multiple traps needed for remote writes. However, the overhead on the receiving side is on a per cell basis, regardless of the approach. This is necessarily the case because cells from a block may be interleaved by unrelated cells from other hosts when they arrive at the destination. Thus, any performance difference is only due to the decreased number of send-side traps, which is small in our implementation. On the other hand, using the block transfer primitive to transfer single cells leads to higher latency because of the overhead of pointer checking inside the kernel.

4.4 Chapter Summary

In this chapter we have described the implementation and performance of the network access model using software emulation on current generation processors and controllers. Because of the simplicity of the model, we are able to provide very high performance, flexibility, and protected user-level network access with minimal overheads. Our implementation experience confirms our belief that it is useful and feasible to support the notion of remote memory on modern workstation clusters. Given next generation processors, with demultiplexed I/O interrupts and special support for emulation code, as found in the DEC Alpha [59], the cost of software emulation could be substantially lowered. Software emulation has a flexibility advantage over a pure hardware approach because application-specific operations can be easily added. Additional performance gains are also possible by redesigning the network to use the memory or cache bus instead, as is done in dedicated multiprocessors such as Alewife [2].

In the final analysis though, the network access model is only an intermediate step. That is, the performance of the network access model is only important in terms of its contribution to the overall performance of applications that use it. In the remainder of the thesis, we discuss the impact of the network access model on distributed programming models and distributed systems.

Chapter 5

APPLICATIONS OF THE NETWORK ACCESS MODEL — RPC

5.1 Introduction

Remote procedure call (RPC) is the established programming model used in modern distributed systems. Building high-performance RPC systems is thus an important aspect of distributed system design.

The purpose of this chapter is two-fold. One goal is to demonstrate that it is possible to build an efficient RPC system using the memory-based network access model described earlier. To this end, we describe the design, implementation, and performance of a prototype RPC system, called RAPID. RAPID is prototyped on the DECstation 5000/200 on the FORE ATM network running the Ultrix operating system. The implementation and performance of the low-level remote memory primitives are exactly as described in Chapter 4.

A second goal for this chapter is to demonstrate that the choice of a network access model can have a dramatic impact on the performance of distributed programming models such as RPC. To this end, we compare the design and performance of RAPID with another RPC system that we built, called FRPC [66]. We designed FRPC about three years ago to study the limitations to latency on high-speed networks like ATM and FDDI compared to traditional networks like Ethernet. FRPC was a highly optimized system built on a byte-stream network access model using the same hardware and operating system software subsequently used to host RAPID.

5.1.1 Structure of RPC Systems

Traditionally, RPC systems have been structured as shown in Figure 5.1a. Clients and servers are placed on different machines; during the run-time binding process, the client imports the interface previously exported by the server. RPC stubs create the illusion of a simple procedural interface for the client and server. Stubs are application-specific and depend on the particular service function that is invoked. Stubs use the RPC transport layer to reliably transfer call arguments and results between the client and the server. Typically, the transport protocol used in RPC systems is a packet exchange mechanism, where a response packet from the server acknowledges a call request from the client, and the next call request from the client acknowledges the previous response from the client.

Conventional RPC design is layered. Even high performance RPC implementations adhere to this layering principle (except, perhaps, to do upcalls from the device level directly to the higher levels) [12, 58, 66]. For example, the stubs view the RPC transport as a provider of reliable packets or linear memory buffers. The stubs translate between user specified entities (integer, records, arrays) and the packets using marshaling code. The transport layer is isolated from the network layer and interacts with it via a simple unreliable packet transmission interface. Reflecting the layered implementation, addressing is done separately at each level, as is input demultiplexing. For example, stubs use handles as communication end points, the transport layer uses transport identifiers or ports as end points, and finally the network layer uses network addresses for communications.

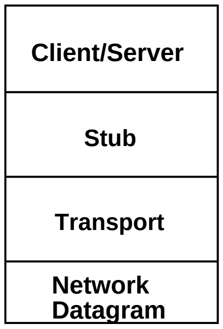


Figure 5.1a: Conventional RPC System

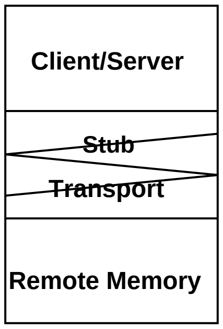


Figure 5.1b: RPC with Remote Memory

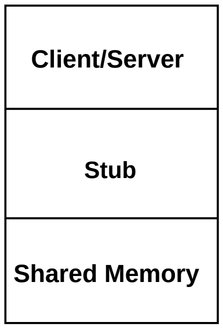


Figure 5.1c: RPC with Shared Memory

What is essential to RPC, however, is not the layering, but rather the simple procedure-call-like syntactic and semantic interface provided to the client and server code. This interface can be supported on top of the remote memory abstraction.

RPC design using the memory-based model is shown in Figure 5.1b. Though layered in principle, there is much more integration between the stubs, the transport and the network (as described in detail in Sections 5.2 and 5.3). In particular, addressing and demultiplexing are done in a uniform fashion (e.g., using <segments, offset> pairs). We view this as an application of the principle of Integrated Layer Processing (ILP) that can be used to structure communication protocols for better performance [19]. Some other examples of ILP are combining data copying and checksum calculations in TCP/IP [17, 75], and exploiting application semantics to avoid extra copies of data for UDP/IP [44].

There are many similarities between the structure of cross-machine RPC using the memory-based network access model and the structure of same-machine RPC on shared memory (shown in Figure 5.1c). Consequently, techniques that yield good performance for same-machine RPCs yield good performance in the cross-machine case as well.

Fast cross-machine RPC systems gain performance advantages by trying to reduce the overhead of marshaling and data copying, protocol processing, and control transfer. The next three subsections describe the design of RAPID in these areas and compare it to other high performance RPC systems that use conventional techniques, e.g., FRPC.

5.2 Marshaling and Data Copying

Stubs are pieces of application-specific code that are generated to bridge the procedural abstraction expected by the client and server and the abstraction provided by the transport layer. Stubs use a process called marshaling to move procedure call arguments and results between the application's procedure variables and the transport layer's memory buffers. Related to the cost of marshaling is the cost of making the transport buffer available to the network in a form suitable for transmission. We refer to collective process of marshaling and moving data between the various layers of the RPC system as "data motion". High speed RPC systems try to minimize the cost of data motion to gain performance.

5.2.1 Data Motion in Conventional Systems

A complete transmission packet contains network and protocol headers, which must be constructed by the operating system, and user-level message text, which is assembled by the application and its stubs. Conventional RPC systems use many strategies for assembling the packet for transmission, with the cost depending on the capability of the controller and the level of protection required in the kernel.

Kernel/User Buffer Mapping

With a controller that does scatter-gather DMA, the data can be first marshaled in host memory by the host (in one or more locations) and then moved over the bus by the controller. In the absence of scatter-gather, the kernel must copy user data an extra time to make it contiguous to the network and protocol headers before DMA can be started. Likewise, given a controller such as FORE's that uses a FIFO, the straightforward technique of marshaling the data into a user-level buffer and having the kernel copy it over to the controller requires an extra copy.

One approach to reducing this cost, for example, the one used in the high-performance Firefly RPC system [58], is to relax kernel/user protection and permanently map all network buffers into user space and allow the user direct access. This is a viable technique if the applications are trusted, or for inter-kernel RPCs.

Kernel Level Marshaling

For general-purpose time sharing systems, though, kernel/user protection is an important consideration. For such systems, there is an alternative technique that retains full protection without incurring the cost of extra copies. This technique, called kernel-level marshaling, was used in the FRPC design.

In an effort to minimize copying of the call arguments, FRPC performs argument marshaling *in the kernel* rather than in the user's address space, as is conventional. To do this, code is synthesized on the fly, which is then linked into the kernel and executed. Code synthesis has been used in the past to generate optimized routines for specific situations to achieve high performance [39, 45]. The focus here is slightly different: it is more concerned with avoiding the copy cost rather than generating extremely efficient code for a special situation.

At bind time, when the client imports the server's interface, the client calls into the kernel with a template of the marshaling procedure. The kernel directly supports simple-valued types such as words, bytes, halfwords, and pointers to bytes. Using the template, the kernel synthesizes a marshaling procedure. In many cases, marshaling is typically simple and involves only assignments and byte copying [7]. Thus, the task of synthesizing a procedure is nothing more than assembling the right sequence of primitive instructions. The marshaling procedure contains code to check the validity of each input argument passed at run time. This approach has the benefit that since the size of the request and reply are known in advance, the more general multi-packet code path can be avoided if arguments and results fit in a single network packet.

The kernel then installs the synthesized procedure as a system call for subsequent use by this specific client. Thus, the stubs linked into the user's address space do not marshal; they merely serve as wrappers to trap into the kernel where the marshaling is done. A client

RPC sees a regular system call interface with all the usual protection rules that go with it. This approach has the benefit of performing the minimum amount of copying required without compromising the safety of a firewall between the user and the kernel, or the user and the network controller. However, this scheme does impose the overhead of probing the validity of pointers before data can be copied. Further, there is a relatively heavyweight trap into the kernel.

On the receive side, especially on the server, it is difficult to fully exploit the benefits of kernel-level marshaling. This is because a server typically exports several procedures with different types of arguments. Thus, it is often unclear which particular marshaling procedure should be invoked. In such cases, a generic unmarshaling procedure is called, which might make it necessary for the server stub to perform additional copies.

5.2.2 Data Motion Using Remote Memory

The cost of data motion in an RPC system is influenced by the network access model. An advantage of the remote memory model, which provides protected, remote memory that can be shared between a client and server, is that it is natural to extend to the cross-machine case the optimization techniques used for high-performance same-machine RPC, such as URPC [8].

Following the URPC approach, in our system the server exports stacks that are then imported by clients at bind time. On an RPC call, the client stub picks an available stack for the server and builds a call frame on that stack using the remote write operation. In the absence of call-by-reference, the call frame is identical to that for a local call. There is really no marshaling or demarshaling per se; data moves directly from source memory to destination memory without unnecessary copying or buffering. (Using writable stacks to simplify demarshaling can be done even without the remote memory model [37], however doing so is more complex.) Once the stack is ready, the client activates the server by writing a flag word in the server, for which the server polls. Call-by-reference is straightforward to provide through the remote read and write primitives. In this case, the references placed on the call frame must contain a segment descriptor and offset into an exported client segment. The segment can be protected with read-only privileges against unauthorized server access if necessary.

The technique of sharing a writable stack is most beneficial when the underlying compiler and processor support separate argument and execution stacks. In this case, only the argument stacks are shared between the client and the server. Contemporary RISC architectures and compilers, however, favor a unified argument and execution stack, which would permit a malicious or incorrect client to crash the server. The execution stack must therefore be protected during server execution. Our implementation allows efficient revocation of segment write access (using the `rr_setprot` call), enabling the server to execute safely. The server exports each stack as a separate segment, write-protecting the segment before an RPC begins. The standard argument checking that is performed by the server is then adequate protection against ill-behaved clients.

It is important to note that data motion is kept to a minimum in the scheme described above: data moves from source memory to destination memory without any extra copies or page remapping overheads. A combination of features makes this possible. First, each network transmission is a write request to a specific location and is therefore self-identifying. In other words, buffering of data before demultiplexing is not required. Second, the implementation takes care to avoid extra memory transfers by moving data from user memory to user registers and thence to the outbound FIFO. In the absence of processor-initiated DMA to the FIFO, this is the best possible scenario.

5.3 Transport

RPC systems have typically relied on reliable packet exchange protocol similar to the one first used at Xerox [12]. The packet exchange protocol is layered directly over network datagrams for communicating within the same physical network, or on top of internetwork datagrams such as UDP/IP for communicating across multiple networks. The performance of an RPC system is affected by the transport mechanism, which in turn is influenced by the access model and the network type. We describe below some aspects of RPC transport design as it relates to network access models.

5.3.1 RPC Packet Exchange

Modern LANs have low loss rates and the function of the transport layer is to provide an efficient mechanism to cope with occasional data loss. Traditional RPC transports were designed for the Ethernet, where typical RPC call arguments and results fit into a single packet (about 1500 bytes). Consequently RPC protocols have been optimized for single packet exchanges [12].

In the standard RPC packet exchange protocol, the client-side transport sends a network packet containing its arguments to the server. (In the common case, RPC arguments occupy less space than what the network packet can support.) In the absence of any errors, the server's response packet containing the call results acknowledges the client's request packet. A subsequent call from the client acknowledges the previous server response packet as well. Thus in the common case—arguments and results fit into single network packets and there are few errors on the network—exactly two packets are exchanged per RPC. In the uncommon case, when multiple network packets are required for sending arguments or returning results, the RPC packet exchange protocol exchanges an acknowledgement per network packet.

With the small cell size of an ATM network, however, data for typical RPC packets may not fit into a single ATM cell. Thus, relying on a packet exchange (in effect, a cell exchange) protocol is not optimal.

One alternative to exchanging cells is to rely on an intermediate ATM adaptation layer (AAL) between the RPC transport and the ATM network layer. Recall from Chapter 4 that the AAL consists of a segmentation and reassembly (SAR) sublayer and a convergence (CS) sublayer that provides the logical abstraction of an unreliable datagram. The AAL accepts RPC transport packets and breaks them up into cell-sized fragments that are individually sent to the destination where they are reassembled into a complete RPC packet. Since AAL does not provide reliability, an aggregate group of cells is treated as one higher-level message with one acknowledgement at the RPC transport layer. Thus, the entire message must be retransmitted even if only one cell is lost. But, if cells are rarely lost, this scheme has the advantage of a straightforward implementation of the RPC transport at the expense of a SAR layer. This was the approach taken in the FRPC design.

However, given the per cell cost of segmentation and reassembly processing, especially in software, performance can be improved by bypassing the SAR and CS sublayers. Notice that remote memory references are self addressing, i.e., they identify where incoming data is to be stored. Thus, reassembly is obviated because the data is simply moved to the right location a cell at a time without being reassembled into an intermediate buffer and then moved, possibly to multiple discontiguous memory locations. Using unreliable remote memory access, there is an alternative scheme that not only avoids the cost of SAR processing but also avoids the cost of per cell acknowledgment that standard RPC packet exchange would incur if used directly on a per cell basis. This scheme is based on the notion of blast protocols and selective acknowledgements [15, 18], and is used in RAPID.

A fixed number of cells (say N) are grouped into an “acknowledgment unit”. These cells are written to the remote site without waiting for an acknowledgement. Since ATM networks guarantee cell sequencing from a particular source, the arrival of the last cell indicates that the previous cells have either arrived or been lost. When the destination detects the Nth cell (or a timeout occurs), it writes a single cell at the source with a bit mask indicating the cells it has received. The source then rewrites only the missing cells, if any.

The only challenge in this scheme is to ensure that the destination can detect whether a cell has been written or not. Since each cell write is guaranteed to be atomic, it is sufficient to detect if a particular word within a cell has been written. To detect if a particular word has been written, the destination has to know in advance what the contents of that word is. Thus the source first writes (to a fixed place in the destination) the contents of a fixed word within each cell of the subsequent N cells. It then writes the N cells before waiting for a write from the destination. Thus in the case where no cells are dropped, every group of N cells requires a constant overhead (2-3 cells) as opposed to the $2N$ cells that would be required if each cell were separately acknowledged.

5.4 Control Transfer

Context switching causes a significant portion of the overhead in RPC [58]; in addition, there is a substantial impact on processor performance due to cache misses after a context switch [49]. An RPC call typically requires four context switches: switching the client out, switching the server in, switching the server out, and finally switching the client back in. Two of these—switching the client or the server out—can be overlapped with the transmission of the packet. Systems with high performance RPC usually have lightweight processes that can be context switched at low cost, but unless there is more work to do in the client and the server, or no work elsewhere, a process context switch usually occurs. The DECstation 5000/200 running Ultrix has context switching times that can be significant compared to the latency of a small packet and it is thus desirable to avoid this cost where possible.

The RAPID implementation is entirely at user level and employs a user-level threads package. To avoid the cost of context switching, we chose to spin wait the threads at user level, relying on kernel time slicing for fairness among processes. This scheme clearly favors latency over throughput, but blocking schemes could be used as well at the expense of increasing the latency by the cost of the context switches. Our implementation of the network access model, as described in Chapter 4, is capable of transferring control to a process on an incoming remote operation. This mechanism can be used instead of spin waiting at the cost of increased latency.

FRPC also used spin waiting to avoid the context switch cost. However, this spin waiting was done inside the kernel and FRPC had to ensure that the spinning did not extend past the client's time quantum if there were jobs in the run queue. The basic spin-waiting scheme used in FRPC can be extended to block the caller without spinning if the expected round-trip is greater than some threshold related to the context switch penalty. An estimate of the round-trip could be obtained either statically using a user supplied hint, or dynamically, by using past response times as an estimate. In general, this technique trades off throughput for latency if there are processes on the run queue waiting their turn.

In contrast, spin waiting at user level is trivial to implement and requires no support from the kernel. Further, the decision to spin wait or block can be driven by application-specific knowledge about the completion time of requests.

To summarize, RAPID implemented on top of the remote memory model has the following key features:

- The tight integration of RPC stubs, transport, and the network to simplify copying and demultiplexing.

- Direct marshaling and demarshaling of data into the target address space as done in URPC.

- A transport based on blast protocols and selective retransmissions instead of the traditional packet exchange.

5.5 Performance

We describe the performance of FRPC and RAPID on DECstations 5000/200s. In both systems, stubs were hand generated for the procedures that we measured and spin waiting was used instead of blocking.

We measured the performance of RPC on two otherwise idle workstations connected by a switchless ATM network. The results are shown in Table 5.1. The column marked **Minus** shows the time required for an RPC involving two integer arguments and the return

Table 5.1: RPC Performance with Different Network Access Models

Cross-Machine RPC Time (μs)		
System	**Minus**	**MaxArg**
RAPID	93	513
FRPC	167	675

of an integer result; the computation time required to calculate the result is negligible, and the data exchanged between the hosts fits in a single ATM cell. In both systems, this RPC takes a fast path and is an indication of the fastest speed possible with the two RPC systems. Notice that the time for RAPID is considerably smaller. The column marked **MaxArg** shows the roundtrip time to make an RPC to the server with 1500 bytes of data and to get an integer result back. It must pointed out that the performance of RAPID for larger data transfers is sensitive to the choice of N, the “acknowledgement unit”. An earlier version of RAPID, with N = 6, gave slightly worse performance than the current version, which uses N = 16.

We attribute the higher performance of RAPID to several factors related to the use of the remote memory model.

First, the use of the memory-based network access model substantially reduces the interaction with the kernel and obviates the use of the ATM device driver. The RAPID RPC transport mechanisms directly perform the low overhead remote writes. In contrast, traditional RPC implementations, like FRPC, will typically require much more interaction with the kernel and the device driver.

Second, RAPID avoids data copying without the complexity of page remapping or kernel-level marshaling used in traditional systems to get good performance. RAPID can avoid this because the network access model supports addressing and demultiplexing very efficiently. This also avoids the cost of going through the SAR layer as well.

Finally, since the network access model allows complete protected user-level access to the network, we avoid kernel context switching overheads by spin-waiting at user level. This avoids the need to interact with the kernel scheduler.

5.6 Chapter Summary

In this chapter we have demonstrated that the memory-based network access model is well-suited for implementing remote procedure call mechanisms, and shown that an RPC system built on this substrate has better performance than an equally aggressive RPC system built on top of a conventional network access model.

However, a more fundamental question we have not yet considered is whether RPC is indeed the correct distributed programming model. While it is the case the RPC has been the model of choice in past and contemporary distributed systems, we will see in the following chapter that there may be environments where it is not beneficial to use cross-machine RPCs. However, there are situations where RPC will continue to be relevant, and in these cases, memory-based network access models and RPC systems designed along the lines of RAPID show considerable promise.

Chapter 6

APPLICATIONS OF THE NETWORK ACCESS MODEL — MULTICOMPUTING ON A WORKSTATION CLUSTER

6.1 Motivation

Recent advances in processor and network performance have increased the attractiveness of parallel computing on workstation clusters used as multicomputers. The processors in current high-end workstations are competitive with the fastest available uniprocessors, and more than competitive with the processors found in tightly-coupled machines. Also, the new generation of local-area networks narrows the gap with the specialized interconnection networks found in more tightly-coupled machines. For example, current ATM networks run at 155 Mbit/sec with 2 Gbit/sec being promised by the mid ’90s, and are scalable in the sense that the aggregate bandwidth can be much greater than the link bandwidth. The wide deployment of these elements and the corresponding economies of scale are likely to have two related effects. First, it becomes possible to use a cluster of preexisting workstations connected by a high-speed network as a cost effective, loosely-coupled multicomputer for running parallel applications. Second, it also becomes increasingly attractive to purchase new commodity workstation and networking gear for dedicated use, rather than a tightly-coupled system. Such loosely-coupled structures built from commodity parts offer advantages over dedicated multicomputers, such as the Intel Paragon [36] and Thinking Machines CM-5 [67]: commodity parts are lower in cost and they can be flexibly scaled and upgraded.

Workstation clusters used for distributed applications rely on heavyweight client/server models and message-based (RPC) communication while parallel applications favor simpler models that involve more direct inter-processor data access. For instance, based on the frequency of calls, and the actual work done by each call, RPC overhead of a few hundreds of microseconds may be considered acceptable. Realistic parallel applications, on the other hand, would require network accesses to take tens of microseconds. Thus, for workstation clusters to facilitate parallel programming, they should support finer-grained, lower-cost network access.

This chapter describes a prototype multicomputer based on a workstation cluster and our initial experience with using it.

6.2 Implementation of a Workstation-Cluster Multicomputer

We implemented our prototype using four DECstation 5000/200s connected by a FORE ASX-100 ATM switch with four ports, each running at 140 Mbit/sec. Before cells can be routed through the switch, virtual circuits have to be established between the two communicating entities. In our multicomputer, between each pair of hosts, we set up a *permanent* virtual circuit (PVC) at boot time. Data destined to several remote memory segments on a single host use the same virtual connection identifier.

Signaling software in the switch allows virtual circuits to reserve a fraction of the link bandwidth. In our prototype, PVCs are set up to reserve 50% of the link bandwidth. We expect this level of bandwidth reservation to provide adequate protection against cell loss due to congestion. In more advanced switches, e.g., AN2 [4], with hardware flow and congestion control mechanisms, bandwidth reservation may not be needed for protection against cell loss.

Given this level of guarantee against cell loss due to congestion and low fiber error rates, it is feasible for the multicomputer to directly use the memory-based network access model as its sole means of communication. Thus, we believe that with little additional mechanism, it is possible to adequately support the needs of parallel applications. Ideally, we do not expect programmers to directly use the network access model to write their parallel applications on the cluster. That is the task of a protocol compiler [29]. This notion is similar in spirit to the idea of using RPC stub generators to hide the details of marshaling from users. In general terms, a protocol compiler analyses a parallel application for communication and computation phases. It then generates communication instructions, e.g., remote reads and writes, at appropriate locations to minimize data and communication overheads.

Our cluster-based multicomputer has good performance and it is useful to compare it with dedicated architectures, such as the CM-5, the Intel Touchstone DELTA, and the Intel Paragon, which is a successor of the DELTA.

Table 6.1 compares the (one-way) latency of sending small amounts of data in the various systems. The average time required to perform a remote write of a given size is shown in the first row. This includes the latency introduced by the switch, which we

Table 6.1: Comparative Communication Latencies in Multicomputers

Latency (μs)		
System	**40 bytes**	**80 bytes**
DECstation 5000/200 ATM	37	55
CM-5 (CMMD)	12	16
Intel Paragon (PUMA OS)	55	55
Intel Touchstone DELTA	71	75

Table 6.2: Comparative Throughputs in Multicomputers

System	**Throughput (Mbytes/s)**
DECstation 5000/200 ATM	4.3
CM-5 (CMMD)	24
Intel Paragon (PUMA OS)	165.0
Intel Touchstone DELTA	10.0

measured to be about 7 μs for each cell. However, this is not a fundamental limitation of ATM switches. We expect that as the technology matures, switching times can be brought down to a few hundred nanoseconds with faster datapaths and virtual cut-through. We did not perform measurements on the CM-5 and the Intel machines, but use values reported elsewhere [43, 69, 73].

Row two shows the one-way cost of sending data to a pre-allocated buffer using the CMMD_scopy routine, which is comparable in functionality to a remote write [69].

Row three represents the performance of send/receive using the facilities of the PUMA operating system [73]. At the time of writing, this represents the best performance reported in the literature. Row four represents send/receive performance reported by Littlefield [43]. Send/receives have an overhead cost for buffer setup and flow control associated with them. The table indicates the performance including the startup cost that is needed for the 3-phase protocol to set up buffers.

Table 6.2 compares the sustained throughput achieved by the various systems. Our implementation overlaps multiple write requests; switch overhead is included in the measurement. Once again, we did not measure the CM-5 and the Intel figures, but quote from the literature.

In relation to the other systems, our implementation has traded off bandwidth for latency.

We chose this bias because we expect applications to use many exchanges of small amounts of data rather than being throughput intensive. However, it is important to recall from Chapter 4 that our implementation achieves about 70% of the throughput that the raw controller hardware is capable of providing.

6.3 Performance of Applications on the Workstation-Cluster Multicomputer

To test the effectiveness of our cluster-based multicomputer and the network access model as a base for building parallel applications, we report on the performance of two parallel programs. These applications were translated from message passing versions that ran on dedicated multicomputers such as the Intel Cube and the CM-5. We did not have a protocol compiler capable of generating remote reads and writes and so the applications were translated by hand.

6.3.1 Ising Model

The first application is an optimization problem that arises in the Ising model, which is used to model the behavior of crystal lattices. Each lattice atom has a characteristic value called “spin”. Atoms interact with their nearest neighbors. Each interaction has an “interaction coefficient”. The lattice as a whole has an energy which is a function of the spins and the interaction coefficient. The objective is to assign spins to the atoms to minimize the total lattice energy. The problem is known to be NP hard, so our particular solution uses a heuristic.

Our program has an optimizer that scans the grid and changes spins if doing so would lower the total energy. The optimizer contains two loops that are written using red/black coloring [54], a common parallelization technique used in scientific applications.

The grid is spatially distributed amongst the processors’ memories such that each processor has two nearest neighbors. The boundaries of the grid contain data that is shared between neighboring processors. During program execution, spin values of atoms that have changed and are on adjacent processors have to be exchanged. The processor that changed the spin writes the new values to the adjacent processor using remote writes. At the end of each sweep through the grid, adjacent processors synchronize with each neighbor using the compare-and-swap primitive.

Table 6.3 indicates the observed speedup of the multi-node algorithm relative to an

Table 6.3: Ising Speedup

Number of Nodes	**Speedup**		
	Grid Size		
	20 X 20	**40 X 40**	**80 X 80**
2	1.5	1.8	2.0
4	1.9	3.0	3.6

otherwise identical single-node solution that uses local memory and no synchronization. For an N X N grid, evenly distributed among P processors, the computation per processor on each sweep is proportional to N^2/P and the data exchanged is proportional to N. For small grid sizes, the time to perform the floating-point computation to calculate the new lattice energy is small relative to the time required to communicate the spin values. Consequently, speedups are modest with increasing processor count. Our measurements indicate that 40X40 is the smallest problem size for which our implementation can be expected to achieve good speedups.

6.3.2 Circuit Simulator

The second application we used was a general-purpose, gate-level circuit simulator that uses a conservative (Chandy-Misra) distributed simulation algorithm. The simulator reads a circuit description and distributes the gates of the circuit among the various nodes. Each gate input is modeled as a queue of events into which output gates write their signal values as the circuit is being simulated.

Each node performs a simple simulation loop. For each gate on the node whose inputs are ready, it calculates the output signal value based on the type of the gate. The output signal values are then fed to the appropriate input gates, some of which might be on a remote node. Most of the parallelism of the application is achieved because the different nodes may proceed independently if their gates have valid inputs.

The simulator is a natural candidate for exploiting remote memory, e.g., the current version has a much simpler communication interface than the message-based version from which it was ported. Gates whose inputs are fed by remote gates are located in an exported segment that is imported by the remote gate. New signal values are simply written by using a remote write.

Table 6.4 shows the results of running the simulator on a 32-bit adder circuit. Perfor-

Table 6.4: Simulator Speedup

Number of Nodes	Speedup	
	Random	**Linear**
2	1.7	1.8
4	3.4	3.5

mance from two distribution of input gates are shown: linear and random. In the linear distribution, with P nodes and N gates, the first N/P gates are given to the first node, the next N/P are placed on the second one, and so on. In the random distribution, the gates were distributed using a random number generator. The particular circuit we used is quite modest and has about 225 gates.

6.4 Chapter Summary

Based on our experience, we believe that the remote read/write model is well suited for building parallel applications. There are two interesting issues to consider here regarding the usefulness of the model.

The first relates to the classic debate between shared-memory and message passing paradigms for programming parallel applications. The network access model that we propose is similar in spirit to shared-memory primitives. It is sometimes argued that typical shared-memory programs, and thus, by extension, applications written using the network access model, may be harder to program than message passing programs due to synchronization constraints. That is, while message passing programs get automatic synchronization as a side effect, shared-memory programs have to explicitly use synchronization primitives. However, despite this seeming disadvantage, many programmers still prefer the shared-memory model because it offers simple semantics and the performance advantage of using only as much synchronization as necessary. (We will return to the issue of synchronization in greater detail in Chapter 7 when we discuss the structure of distributed systems using the memory-based network access model.) Further, with mechanical aids like protocol compilers, the distinction between shared-memory and message passing primitives might be less important in the future.

The other issue is whether the primitives provided by our model are inferior in some ways to the primitives in shared-memory machines. Our prototype has some features that could be viewed as inconvenient compared to dedicated shared-memory architectures. First, our prototype, in contrast to true shared-memory machines, provides remote memory access instructions that are slightly different from those used for local memory. This is a clear tradeoff that allows us to use commodity parts at low cost and with high flexibility. In contrast, machines like SHRIMP [1] provide uniform local and remote access instructions, but at the expense of customized hardware [13]. Second, our model does not provide automatic memory coherence as most shared-memory machines do. Once again, we believe this is a reasonable tradeoff of performance for functionality. Rather than mandate coherence that would be difficult to efficiently support on commodity workstation clusters, we provide users high performance instructions to keep memory consistent at the application level.

At present, our experience with the scaling properties of our workstation cluster and the network access model is limited. Remote reads and writes are point-to-point and thus communication costs could grow quadratically with the number of nodes. However, relative to message passing, which is also point-to-point, we expect remote reads and writes to scale equally well.

[1] Strictly speaking, SHRIMP is not a shared-memory machine, but it has many similar features.

Chapter 7

RESTRUCTURING SERVICES IN A DISTRIBUTED SYSTEM

As we have mentioned earlier, the hardware base for distributed systems has changed significantly over the last decade. Advances in processor architecture and technology have resulted in workstations in the 100+ MIPS range. As well, newer local-area networks such as ATM [48] promise a ten- to hundred-fold increase in throughput, much reduced latency, greater scalability, and greatly increased reliability, when compared to current LANs such as Ethernet.

Previous chapters have considered a new network access model tuned for next-generation networks. We have also shown that this model can be used in two ways: (1) to support parallel programming on local-area workstation clusters using techniques such as protocol compilers, and (2) to build efficient RPC systems to support distributed applications. In other words, though the technology had changed dramatically, we were considering traditional system and application structures on top of a new communication model.

In this chapter we make a more radical departure, and consider a novel organization of distributed systems made possible by the memory-based network model and the new technology. It is our contention that new network and processor technologies will permit tighter coupling of distributed systems at the hardware level, and distributed system structure should change as a result in order to benefit from that tighter coupling.

Distributed systems are typically structured as clients and servers that communicate using RPC- or message-based communication. Such RPC-based client/server systems are highly tuned and relatively efficient for current-generation networks, which are relatively slow, relatively unreliable, and permit only a loose coupling between distributed components. However, such protocols and structures may well be sub-optimal for next-generation networks, for which these assumptions no longer hold.

The rest of the chapter is organized as follows. We first describe the structure of RPC-based distributed systems and the limitations of this structure in newer LAN environments. Next, we discuss the alternative structure for distributed services. The alternative structure is based on the underlying network access model described in Chapter 3. We present data from a simple name server application and from measurements of NFS [56].

7.1 The Trouble with RPC

RPC is the predominant communication mechanism between the components of contemporary distributed systems[1]. For this reason, an enormous amount of energy has been devoted to increasing its performance [37, 58, 66, 70]. Still, RPC times are substantial compared to the raw hardware speed. While this cost is due in part to the latency of network controllers and the software protocols used for network transfer, it is due as well to the semantics of RPC.

RPC performs two conceptually simple functions:

- It transfers *data* between a client's address space to a server's address space. Depending on the implementation, data transfer may be costly due to the (sometimes overly general) stubs required to marshal and unmarshal parameters for transmission, and due to the (sometimes repeated) copying of data between the client or server and the wire.

- It transfers *control* from a client thread to a server thread and back. The work to perform this control transfer involves at least: (1) blocking the client's thread and rescheduling the client's processor, (2) processing the RPC message packet in the destination operating system, (3) scheduling, dispatching, and executing the server thread, (4) rescheduling the server's processor on return by the server thread, (5) processing the reply packet on the client's operating system, and (6) scheduling and resuming the original client thread.

These functions of data transfer and control transfer are fundamentally bound together in the RPC model. Thus, to copy just one byte of data from client to server, an RPC system must perform the control transfer (e.g., the thread scheduling and management in steps 1, 3, 4, and 6 above) as well, which has little to do with the transfer of that byte. Even in high-performance RPC systems, control transfer can take a substantial amount of time. For example, measurements of Firefly RPC, a highly optimized system, showed that control transfer was responsible for 17 percent of the overall time of an RPC with no arguments

[1] In this context, we consider RPC and message passing to be essentially identical.

Table 7.1: Summary of NFS RPC Activity

Activity	**Number of Calls**	**Percentage of Total**
Get File Attribute	8960671	31
Lookup File Name	8840866	31
Read File Data	4478036	16
Null Ping Call	3602730	12
Read Symbolic Link	1628256	6
Read Directory Contents	981345	3
Read File System Stats.	149142	0.5
Write File Data	109712	0.4
Other	109986	0.3
Total	28860744	100

and no results, and 7 percent of the overall time for a call with no arguments and a 1440 byte result [58]. Spin waiting, e.g., as done in our prototype FRPC and RAPID systems, may not be a feasible option in many environments.

Given this problem, we should ask whether distributed applications require a single primitive that unifies data and control transfer. To examine this question for one application, we consider NFS, which is probably the most common example of a distributed service in daily use. We instrumented and measured the primary NFS file server for a collection of 80-100 workstations in our department. Most of our workstations have local disks where individual user files are stored. The file server exports X-terminal fonts, source trees for systems like the Ultrix kernel and GNU distribution, and the `/usr` partition containing executable binaries, in addition to hosting a small complement of users. The exported partitions have a mix of read-only and read-write files, but with a relatively higher proportion of read-only files. Table 7.1 shows an analysis of operations performed at the server over the course of several days. It is significant to note that for the most part (i.e., for all rows except the "Null Ping"), the goal of the RPCs in Table 7.1 is to transfer data—either file data or file metadata—between the server and the client. If that data could be transferred *directly* between the server and the client, then we could avoid control transfers. Thus, ignoring the issues of synchronization for the moment, these RPCs potentially could be replaced with simpler mechanisms that involve only transfer of data, *if* the system were structured in a way that facilitates such transfers.

Using an RPC scheme to perform simple data transfers has two bad effects. First, there

Table 7.2: Breakdown of NFS RPC Traffic

Activity	**Control Traffic (MBytes)**	**Data Traffic (MBytes)**	**Ratio of Control to Data**
Get File Attribute	214	481	0.44
Lookup File Name	246	761	0.32
Read File Data	185	2176	0.09
Read Symbolic Link	59	19	3.10
Read Directory Contents	54	1862	0.03
Read File System Stats.	4	3	1.33
Write File Data	4	271	0.01
Overall Total	766	5573	0.14

is the overhead of scheduling and procedure invocation. Further, RPC-style communication imposes a secondary overhead, because it creates unnecessary network traffic in addition to the actual data bytes being transferred. This additional data that is transferred due to RPC semantics can be a non-trivial fraction of the total data exchanged. We illustrate this phenomenon by considering the network traffic between an NFS client and the server. We classify the activity into "data traffic" and "control traffic". Data traffic represents the data that is *required* by the particular distributed file system protocol. That is, if there was a communication primitive that permitted direct and protected data transfers from server memory to client memory, this is the amount of data that NFS would have to transmit. Control traffic represents additional data that is transmitted because NFS uses RPC as the communication primitive. This classification is significant, because it identifies the amount of traffic that can be eliminated by avoiding an RPC or message passing style of communication with its integrated transfer of control and data.

Table 7.2 shows the breakdown of the data and control portions of the client/server traffic for the snapshot shown previously. Note that this is a cumulative picture of the system after it has been running for a long time. We have not included the overhead of network protocol specific headers. Overall, the control traffic due to the RPC model is about 12% of the total.

An alternative model that reduced control traffic would, in turn, reduce *processing* on the server side. Most interactions between the client and the server involve only data accesses that should not require much server involvement. If we can eliminate both the traffic and the server involvement, we have the potential to improve scalability by lowering both network and server load. That potential can be realized if we use the memory-based network access model and a new structure for distributed applications.

7.2 An Alternative Structure

In this section we describe an alternative structure for distributed applications. We briefly outline the goals of our design and the environment in which we expect to run. We then describe the structure of distributed applications using our memory-based model of network access.

Simply stated, our objective is to improve the performance and scalability of distributed services. To accomplish this, we consider three important design goals. First, the communication system should avoid bottlenecks that might hurt service request times. Second, the load on server machines should be minimized, using client machines for processing where possible, in order to improve scalability. Third, the load on the shared network should be minimized, again to improve scalability. Modern networks using switched point-to-point links can tolerate loads better than bus-based networks like Ethernet, because multiple links can be aggregated between nodes to provide increased capacity when required. However, loading at switches is a potential performance problem that we would like to reduce.

Our design is based on certain assumptions about the underlying workstation and network environment that we expect to use. We wish to build tightly-integrated distributed system clusters, consisting of a modest number of high-performance workstations communicating within a single LAN-connected administrative domain. Newer LAN technologies include hardware flow-control and bandwidth reservation schemes that can guarantee that data packets are delivered reliably [4, 5]. We therefore feel justified in treating data loss within the cluster as an extremely rare occurrence, and regard it as a catastrophic event. This permits the use of simplified communication primitives, such as our simple read/write primitives; a request-response protocol, e.g., RPC packet-exchange, is not needed for reliability and need be used only if a response is required by the sender.

7.2.1 Structuring Applications

The objective of our structure is to separate control transfer and data transfer in a distributed service, in order to remove superfluous cross-machine control transfers, while optimizing data movement. We use the network access model described in previous chapters to effect this separation. Our structure has four important components:

- *Clients and Servers.* The system is structured as clients and servers, as in existing distributed systems. Clients and servers exist on different machines within the cluster, connected by a high-speed local-area network.

- *Specialized Data Transfer Mechanism.* We use the specialized communication primitives described in previous chapters to support direct, protected, remote memory access.

- *Server Clerks.* Each distributed service has server clerks that execute on the *client* machines. All client-server interactions are done through *local cross-address space* communication between the client and the server clerk. Server clerks do not trust their clients; however server clerks and servers are considered part of the same service, and trust each other.

- *Clerk-to-Server Data Transfer.* Clerks and servers can cache data if necessary. Communication might be necessary between clerks and the server to keep the caches consistent. Whenever possible, this communication is done using the remote read/write data transfer mechanism.

Figure 7.1 shows the organization of a distributed service along these lines. Notice that for the most part, control transfers are restricted between a client and its server-clerk. That is, control transfers are primarily intra-node cross-domain calls, which have been shown to be amenable to high-performance implementation [7, 42]. Notice also that our organization maintains the firewalls between untrusted clients and services and the abstractional convenience of procedural interfaces, without relying on conventional mechanisms like RPC for cross-machine communication.

There are obviously many possible variations to the scheme shown, which we do not discuss explicitly; for example, in some cases it might be possible to eliminate the server completely and have the state maintained by the clerks alone.

Our structure bears resemblance to a traditional distributed system that does caching, however, it has some distinctive features. First, we use local caching to reduce cross-machine communication (this is similar to earlier systems). Second, we specifically eliminate cross-machine control transfers in many cases where the clerk can satisfy the client

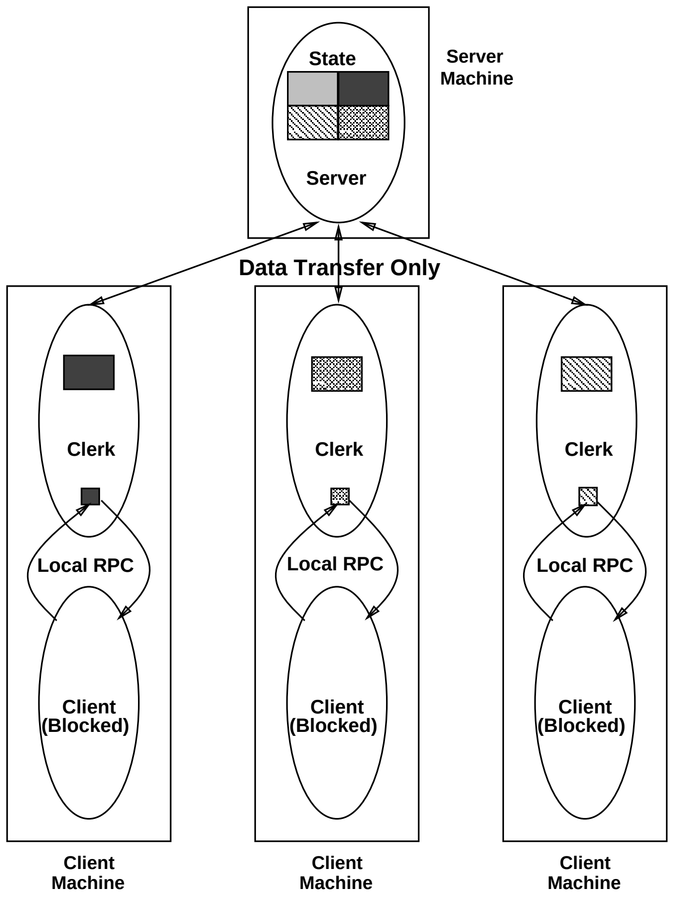


Figure 7.1: Structuring a Distributed Service Without RPC

request by data transfer only. Finally, we simplify data-only communication in both directions; that is, it is possible for the server to eagerly update data on its client-side clerk, or for the clerk to eagerly push data to or pull data from the server.

Our use of clerks has much in common with earlier systems, e.g., Grapevine [11]. For example, the client is not aware that it is talking to a server's clerk. It sees the same abstract RPC interface, albeit local RPC. However, there are some important differences in our use of clerks. Critically, in traditional organizations, clerks communicate with servers using cross-machine RPC, in contrast to our organization, which explicitly avoids its use where possible. In our design, clerks and servers have direct (remote) access to each other's memory and are consequently more tightly integrated with each other.

In our system, the state of the server is distributed and cached locally. Thus, there is a cache coherence and consistency issue that needs to be addressed if client requests are to be correctly satisfied. While the details of the coherence protocol are not significant, it should be noted that we do not require a broadcast capability from the network or the communication mechanism. We do not believe there is one single cache coherence policy that is appropriate for all services. Our structure permits an individual service to use schemes that are most convenient for that service.

In the usual case, we expect the communication between clerks and the server to involve only one-way transfer of data. Since there is no transfer of control to generate responses or acknowledgments on the server, we expect server load to decrease relative to a request-response model.

7.2.2 Locating Remote Data

A key feature of our organization is that the parts of the system that are distributed across the network are parts of the same application. Thus, we can exploit application-specific information to optimize the cross-machine transfer of data and control. For example, the server and server-clerk understand the organization of each other's data structures: a server-clerk can read and possibly modify the server's data structures, and vice versa. (In Section 7.3.1 we show a particular organization of data that allows this kind of access.) This is in contrast to traditional organizations, which typically cannot exploit such knowledge, because the distributed components are different applications separated from each other through message exchanges.

7.2.3 Synchronization

The system structure we propose has several options for handling synchronization issues that commonly arise in building distributed systems.

The first option is that in certain situations we can do without synchronization at all. For instance, consider the case of load balancing in a workstation cluster. Each workstation could update a shared variable with its current load using remote writes. Other workstations would read this value and take appropriate load balancing actions. In this situation, the exact value of the load seen by a reader is not of significant concern, because it is being used as a hint. A similar solution applies in other situations in distributed systems where hints are used.

Often, hints alone will not suffice. In some of these cases, one can exploit certain atomicity properties of the communication model for achieving synchronization. For example, we mentioned earlier that single-word local memory access are atomic with respect to remote memory accesses. This property can be used to ensure, for example, that a flag word in a record is atomically updated. This allows a sufficient level of synchronization in cases where there is single writer and multiple readers. In fact, we use precisely this mechanism in the implementation of the name server described in Section 7.3.1.

A third option is to use the synchronization provided by the `CAS` operation supported by the network access model. This primitive is sufficiently powerful to build higher level synchronization primitives.

A final option is to use the control transfer option supported by the network access model to implement RPC-like synchronization semantics. In other words, by providing a mechanism to implement RPC-like behavior, we allow applications to use the implicit synchronization inherent in RPC. We should note in passing that, traditional organizations, with their exclusive reliance on RPC semantics, offer only a single option for synchronization. Unfortunately, this option has performance disadvantages that cannot be avoided in current systems.

7.2.4 Security

In the new organization, clients access the services of the clerk using local RPC, which maintains complete protection firewalls. Thus, clients cannot damage servers or their clerks.

Further, our network access model based on remote segments is secure under the assumption that the kernels and privileged users on each machine are trusted. With this assumption, it possible for servers and server clerks, which execute on different machines, to trust each other. This might appear to be a security flaw in the design, but it need not be so. For instance, many environments routinely share files through NFS servers with exactly the same guarantees.

However, there are environments where such trust may not be warranted. Distributed systems running in these environments have traditionally used encryption techniques to ensure authentication and security [10, 63]. The underpinning for such schemes is that data is encrypted and decrypted using secret or public key schemes. The choice of the particular communication primitive—RPC or remote memory—is irrelevant. With the memory-based access model, this implies that each read and write has to be encrypted and decrypted. The software emulation technique that we use in our implementation will not provide adequate performance in this case. However, it is feasible to do encryption and decryption in hardware. In fact, the AN1 controller [57] has mechanisms to decrypt and encrypt data using different keys as data is transmitted or received. Thus, we expect that with suitable hardware support, the network access model and the structure that we propose can be used in a secure fashion.

7.2.5 Heterogeneity

It might appear that the proposed structure and the memory-based network access model rely heavily on the existence of a set of homogeneous machines. In fact, this need not be the case. The most common case of heterogeneity— different byte orders—is straightforward to accommodate as a simple extension to the current implementation of the network access model. Recall that since we use programmed I/O to move data between the controller FIFO and memory, byte swapping can be readily performed. This scheme requires a bit in each incoming request to decide whether to swap or not. Even with a hardware implementation of the model, this functionality can be provided. In fact, even very early network controllers, such as the Ethernet LANCE, have a facility to swap bytes during data transfers between host memory and the network [1].

Other kinds of heterogeneity, such as different word sizes or different floating point formats, are more difficult to deal with. We expect that some form of presentation conversion, as found in RPC stubs, will be required before applications can access the data sent from a heterogeneous machine.

7.2.6 Cache Consistency and Recoverability

Our structure encourages local caching of data and state, and thus cache consistency mechanisms and recoverability from machine crashes are important concerns. However, these issues are independent of whether we use our structure or a traditional RPC-based structure. For example, many file system designs, e.g., Sprite [52], SNFS [62], and AFS [34], which use RPC, require mechanisms for recoverability and cache maintenance.

In traditional RPC-based distributed systems, the RPC runtime and transport implement timeout and exception mechanisms to automatically notify the user of remote machine failures. It might appear that in our organization, fault-tolerance might be a difficult goal to achieve. In fact, while the read/write primitives by themselves do not provide fault-tolerance, they can be used by a language or runtime system to provide notifications of remote machine failures.

The key distinction between RPC-based services and our organization is that RPC-based systems integrate fault-tolerance with data and control transfer. In our organization, the communication primitives by themselves do not provide fault tolerance, but they can used with timeouts to provide the required level of failure protection. For example, a service that required fault tolerance could implement a periodic remote read request of a known (or monotonically increasing) value. Failure to read the value within a timeout period can be used to raise an exception. Notice also that in both approaches, the fundamental mechanism needed for failure detection is timeouts. Thus, as long as the system supports efficient timeout mechanisms, we expect comparable functionality can be achieved with either approach.

7.3 Example: A Simple Name Server

To gain experience with our restructuring strategy we performed two studies. In the first, we implemented a name server that is relatively simple, but which exposed us to some of the implications of the new structure. In the second study, we estimated the performance improvement of minimizing control transfer in a traditional distributed file system. This section describes the first of these experiments, and the lessons we learned from it. The following section describes the second study.

For the future, we expect distributed systems based on our structure to run on workstation clusters on highly reliable networks. For the purposes of our experiments, we used a testbed that behaved like our target environment. Our testbed consists of a pair of DECstations connected to a switchless ATM network. Being a private, isolated network, the environment is practically error free and meets our assumptions concerning the processor and network environment.

7.3.1 Design of the Simple Name Server

Recall from Chapters 3 and 4 that our communication primitives rely on named segments. Users *export* segments by name for other users to subsequently *import*. An owner of the segment may also *revoke* a segment (i.e., make it unavailable). The purpose of the name server is to maintain the registry of segment names and information so that importers and exporters can communicate.

The name server is logically structured as a centralized service, but it is physically organized as a distributed collection of clerks, one per machine. Unlike the organization shown in Figure 7.1, there is no centralized server. Communication between the clerks implementing the distributed name service is done using the remote access primitives themselves. Certain well-known segment names have been reserved on each machine to allow the name service to bootstrap itself.

The name server implements procedures to add information about exported segment names, to lookup name information, and to delete names. The name server is trusted and privileged; its clients are not ordinary users but kernels. The relationship between a user exporting or importing a name, the kernel, and the name server is described below.

A user exports a segment by name by making a kernel call, which is turned by the kernel into an `ADDNAME` RPC to the name service. This RPC is serviced by the local clerk of the name service, which enters the relevant information into its name registry. A segment import by a user results in a kernel call, which in turn makes a `LOOKUPNAME` RPC to the name service. Similarly, segment names that are deleted by users result in a `DELETENAME` RPC.

Each time a segment is exported, the kernel assigns it a monotonically increasing generation number. Generation numbers accompanying each remote request allow the kernel to disallow operations on stale segments. Generation numbers allow a simple implementation of the delete operation within the clerk. A delete operation merely marks the entry invalid in the local cache. There are sufficient bits in the generation number so that, even under heavy load, it wraps around slowly enough to allow name clerks considerable latitude in propagating deletions.

Name additions are handled slightly differently from deletions. One option we considered was for the clerk on the exporter's machine to propagate the name to each of the other clerks. Thus, a subsequent `LOOKUPNAME` RPC can be satisfied by a simple local lookup. Unfortunately, this can limit the scale of the name service, because it involves writes to all remote machines, most of which may not require the name in any case. A better option, and the one we have implemented, is for the clerk on the importing machine to do a remote read operation to retrieve the name.

A name server clerk periodically refreshes its cache of imported names. At the end of each refresh operation, imported entries that are no longer valid are purged from the name cache and from the kernel's tables. Thus, a lookup operation after a cache refresh will cause a remote read to get the most up-to-date information. Also, after a refresh (but not necessarily before), remote operations using stale entries will fail locally at the source allowing the source a chance to recover. Further, the source can timeout on a read that uses stale information and redo the import operation. In addition to these mechanisms for coping with stale name entries, users can force a specific import operation to do an explicit remote lookup.

7.3.2 Implementation

Name clerks are created at boot time. When each name clerk is started, it exports a well-known segment granting write privileges to other clerks. The kernel treats this export operation as special and uses it to update the in-kernel export segment table against which incoming remote operations are checked. After a name server clerk has exported its segment, it imports the well-known segment from each of the other machines on which it expects to do lookup operations.. Once again, this import operation is simply for the kernel to update its information of imported segments.

Each clerk's well-known exported segment is used as a registry to hold information about other named segments. The registry is organized as an open-addressed hash table. When a user exports a segment, the information about the segment is hashed by name into the clerk's hash table. Each clerk uses the same hash function. Thus, unless there are collisions, information about a particular name will be in the same position on all the clerks. This is a convenient performance optimization as described below.

In response to an `ADDNAME` RPC from the kernel, the clerk adds the information about the exported segment into its hash table using local memory operations. Subsequently, a remote importer, via a kernel call, can contact its local clerk, and present a name for retrieval. The clerk first checks in its local table for the appropriate name. If it finds it, the information is returned to the kernel, which updates its tables and returns a handle to the user who made the import call. If the clerk's local lookup fails, it uses a user-supplied hint, specifying a remote machine, to perform a read operation on the appropriate remote clerk's well-known segment. It is in this situation that using identical hash functions on all clerks pays off. Identical hash functions allow the importer's clerk to locate the name usually with a single remote read operation. Sometimes this will fail, for example, if the remote clerk encountered a hash collision when originally inserting the name and rehashed the name to some other hash bucket. If the result of the read operation does not return the appropriate name, the local clerk has three options: (1) keep probing different hash location using remote reads until the record is located or the hash table is exhausted, (2) use a remote write with control transfer to request the other side to check its name table, and (3) probe a few times and then transfer control. The choice of which option to use is application dependent and is related to the cost of doing lookups, the number of expected lookups, and the cost of transferring control. Given the relative costs of remote data transfer in our implementation, we use the first option, because that gives us the best performance. Control transfer is a viable option in our case only if we expect seven or more collisions to occur in the hash table.

In the case of this particular name server example, the organization follows the ideal model described in earlier sections. Thus, all remote communications involve only transfer of data. Communication between the user, the kernel, and the clerk involves only local transfer of data and control. Since Ultrix has no efficient same-machine RPC mechanism, we use the existing system call mechanism to transfer data and control locally. Initially, the name clerk makes a special system call that blocks it in the kernel. When a user request is made, the kernel blocks the user and unblocks the clerk's call. The clerk performs the requested function and calls back into the kernel, which uses the call arguments as the return result of the original user request.

We should mention that the name server described here is a low level service. Its only responsibility is to help the kernel manage the export and import of segment name information. We have therefore used a set of coherency mechanisms, such as generation numbers, periodic cache flushes, and kernel-supplied hints, that are appropriate to this service. Undoubtedly, other kinds of name services may require different coherency guarantees. By implementing the segment name server, we gained experience with locating information, synchronization, and organizing clerks in the new structure.

7.3.3 Performance

Table 7.3 shows the performance seen by the user for exporting, importing, and deleting a segment. On all three operations, the kernel mediates between the name server and the user. Notice that the difference in time (68 μs) to perform a lookup when the data is available locally and when it is not, is comparable to the cost of a remote read operation (45 μs) from Table 4.2. That is, cross-machine communication cost is basically the cost of simple data transfer. The information that is retrieved on a lookup operation fits in a single ATM cell. With improved same machine communication performance, we expect the overall

Table 7.3: Name Server Performance

Elapsed Time (μs)		
Operation	**Cached**	**Uncached**
Export (`ADDNAME`)	665	N/A
Import (`LOOKUP`)	196	264
Revoke (`DELETENAME`)	307	N/A
`LOOKUP` with notification	524	

performance to improve even further.

The last row represents the cost of doing a lookup operation with control transfer. In this case, the importing clerk performs a remote write with notification. The write contains arguments for the lookup operation including a pointer back to the importer's exported memory segment. Upon notification, the exporting clerk uses the arguments to do the lookup. The results of the lookup are directly written to the importing clerk's memory using a remote write. In the meantime, the importing clerk spin waits (at user level) for the data to arrive. Another option, though correspondingly more expensive, is to not spin wait the importer but instead have the exporter use a remote write with notification.

7.4 Example: A Distributed File Service

Distributed file systems, such as NFS, are perhaps the most common examples of distributed services in daily use. Most distributed file systems are implemented using RPC-based clients and servers. However, much of the traffic in a file system need only involve data transfer; the control transfers are sometimes an unneeded cost imposed by the use of RPC between clients and servers. Recall from Section 7.1 that the benefits of eliminating control transfer are: (1) lowered overheads due to context switching, blocking, and procedure invocation, and (2) eliminating the processing overhead for unnecessary data traffic. In this section, we analytically evaluate the impact of the new structure on the performance of a distributed file system service using a functional model of the system.

7.4.1 The Distributed File System Model

We assume that the client machine runs a server clerk and that the clerk and the server cache data. This is fairly general model and even encompasses systems such as traditional

NFS (where the client kernel acts as the clerk for the remote server). Since we use caches on clients and servers, it is necessary to have a policy for cache-consistency. However, we are not directly concerned with the particular choice of protocol that is used and our system model does not implement one. Many coherency protocols are well known and are in use in current distributed file systems, e.g., the Spritely NFS system [62], Sprite [52], and AFS [34] all use cache-consistency schemes. (Even NFS has a cache-consistency policy, albeit a weaker one.) Traditional multiprocessor cache-coherency schemes can also be used, as suggested by the work on the xFS protocol [21].

Although our file system model does not explicitly account for coherency traffic, we believe coherency schemes can be built using our communication primitives and our file system model. For example, workstation-cluster file system designs such as Calypso [50] use an RPC-based distributed token management scheme to handle cache coherence. This scheme can be extended to use our communication primitives without involving control transfers in most cases. Token acquire and release can be implemented using compare-and-swap operations. Token revocation is trickier. One option is to use control transfer (e.g., using Hybrid-1 as described below); another is to delay revocation during certain conditions, as is done in Calypso, which can be done without control transfer. For the commonly occurring sharing patterns in distributed file systems, we expect the usage of control transfer for coherence to be rare.

Our system model organizes the cache into different distinct areas, each containing different types of information organized as shown below. This organization, allows us to exploit pure data transport mechanisms without the penalty of control transfer.

File Data This is the traditional file buffer cache that caches regular file data and forms the bulk of the cache. Data within the cache can be located using a file handle and block number within the file.

Name Lookup Data This area contains information to translate file names to file handles. Most conventional systems have a separate name cache that serves this purpose.

File Attributes This cache area contains file attributes such as creation time, file size, etc. Entries in this cache can be retrieved given a file handle.

Directory Entries We keep the contents of directories in a specifically designated area of the cache. This allows fast directory searches. From measurements on our departmental file server, which is typical, we observed that the entire directory contents of the server could be cached with about 2.5 Mbytes of data. With an additional 40 Kbytes of memory, even symbolic link information on our server can be completely cached. Caching this information is helpful, because reading symbolic links and directory entries accounts for about 8% of the activity in Table 7.1.

Our model of the file service is simple but captures the performance effect of separating control transfer from data transfer. Our underlying communication primitives allow several alternatives to coordinate the movement of data between server and clerk caches. We consider four of these alternatives below:

Write Requests Only The first alternative, and the simplest, is for the source of the data (server or clerk) to supply data to the destination using remote writes with no notifications at all.

Read Requests Only The second alternative is for the eventual destination of the data to fetch the data from the source. This was the method of choice in the name server of the previous section. In the current experiment, we use block read requests to fetch data from the server.

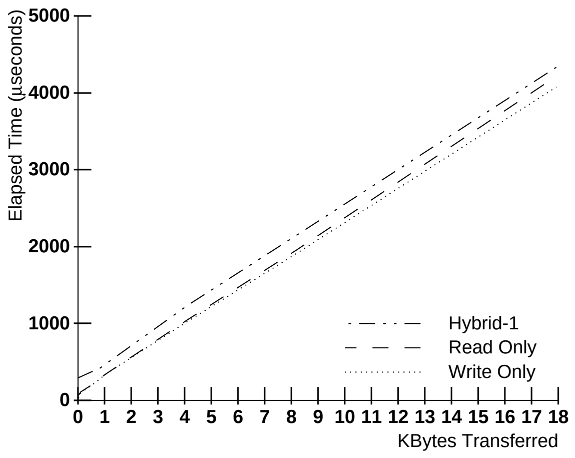
Hybrid-1 Unlike the previous two schemes, which are pure data transfer schemes, this scheme uses a single write request with notification, followed by one or more return write requests. This was one of the alternatives we considered in the name server example. The destination makes a write request (with notification) describing the data transfer parameters. The source then performs one or more return write requests back to the destination.

Hybrid-2 This is the same as Hybrid-1 above except that the source sets the notification bit on the last write request to the destination.

7.4.2 Performance

We show below the quantitative impact of separating control and data transfer on file system operations. First, as a simple metric, we measured the transfer times for various data transfer sizes, for the four data and control transfer alternatives described above. The measurements were done on DECstation 5000/200s connected to a private ATM network. Figures 7.2a and 7.2b show the performance of the various alternatives. For comparison, we show the performance of the native RPC currently used by distributed services within Ultrix. The Y-axis shows the elapsed time for transferring data, excluding any other clerk or server activity and assuming a perfect hit rate in the remote cache. The X-axis shows the amount of data transferred. Figure 7.2a shows overall performance and Figure 7.2b shows the performance for the small- and medium-sized cases.

The performance figures for the RPC system include the cost of marshaling and demarshaling data. Native Ultrix RPC uses UDP/IP as its underlying transport. Since the Ultrix UDP implementation does not permit single RPCs larger than 8 Kbytes, we measured the large byte cases using multiple RPCs as needed.



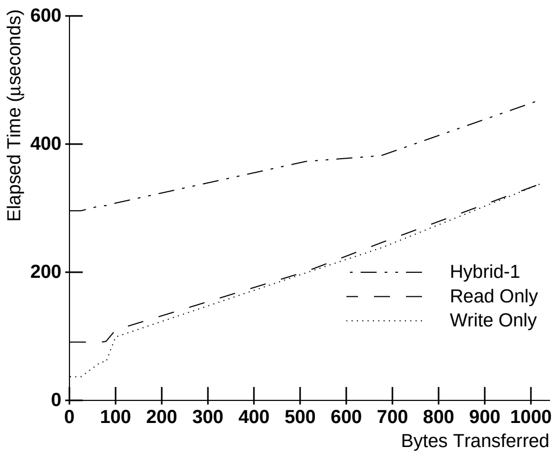


Figure 7.2a: Overall Comparative Performance of Communication Mechanisms

Figure 7.2b: Comparative Performance of Communication Mechanisms for Small Packets

The graphs show that the remote memory schemes have substantially better performance than Ultrix RPC. Even the two hybrid schemes, which involve limited control transfer, outperform RPC. However, the performance of Ultrix RPC must be taken in context, because there are many finely-tuned RPC systems that have better performance that Ultrix RPC.

To evaluate the impact of separating data and control transfer, we compare the performance of two alternative structures for a distributed file system. From the perspective of a distributed system, we are interested in two performance metrics: (1) the total latency seen by a client and (2) the load on the server.

We assume both alternatives cache data locally. However, in the first alternative, the file system uses a fast RPC-like cross-machine mechanism, viz., Hybrid-1 (the faster of the two hybrid schemes that transfer control and data). This is similar in spirit to the design of traditional RPC-based systems, e.g., NFS.

The second alternative uses the proposed new structure and relies primarily on a pure data transfer scheme. That is, the clerks and the server directly access each other's caches using remote reads and writes, just as was done in the name server example of the previous section. If there is a miss in the remote cache, control is transferred to the remote process, where a procedure is activated to locate the missing data and write it back to the source using remote write operations.

We are assuming in our model of the file service that the caches (as described in Section 7.4.1) are organized as several hash tables in a fashion similar to the name service. Thus, we expect synchronized access to data to be implemented analogously to the name server. That is, a file system clerk would perform one (or more) remote reads to fetch a block of data or metadata. A flag word in the block would indicate if the data is valid or not. The atomicity of remote access guarantees this. If the data is valid, then a comparison of the block number shows if there was a miss or not. Thus, we expect only minimal overhead (a few compare and branches) for proper synchronization and miss *detection*. We have therefore ignored this cost in the comparative measurements below.

For the sake of concreteness, we assume that the file system presents an interface similar to NFS, i.e., it implements operations like those shown earlier in Table 7.1.

Since both the schemes that we compare cache data locally, the performance of client requests that hit in the local cache would be similar in both. However, the performance of client requests that miss in the local cache could be different. In the first scheme, using Hybrid-1, client latency is affected by (1) the time to send the request and control information to the server, (2) the processing time on the server, and (3) the time to write the results back to the client. To evaluate these components, we directly measured the cost of all three for each file system operation. Items (1) and (3) were measured from our implementation of Hybrid-1. To estimate the processing time at the server, we measured the processing times on an actual NFS server with warm caches on an isolated ATM network.

Ultrix RPC and marshaling costs are not included in this measurement. In the second scheme, the latency seen by the client is dependent primarily on the low-level cost of emulating the remote memory operations, since there is no server involvement.

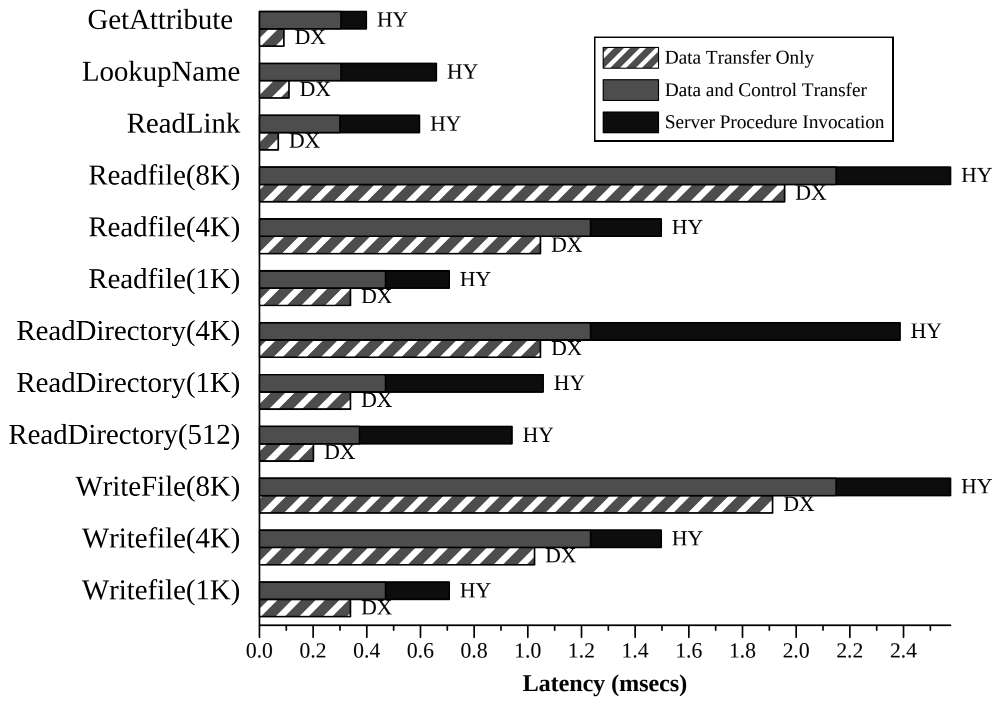


Figure 7.2: Request Processing Latency Seen by Client

Figure 7.2 compares the performance of the two schemes for representative file operations. The measurements were done on DECstation 5000/200s connected to a private ATM network. We assume 100% hit rates in the server cache. We also neglect the communication cost between client and clerk. Thus, these are best-case figures. Note however, that if there is a miss in the server cache, overall performance will be dependent on the disk transfer time rather than differences in the structure of the service. For each file operation, we show the total latency for that operation implemented using Hybrid-1 (HY), and the total latency for the pure data-transfer scheme (DX). In the case of Hybrid-1, the latency includes two components: the time to transfer data and control, and the server processing time. In the case of pure data transfer, the entire latency is due to the data transfer primitives. We must point out that in the pure data transfer scheme, we have ignored the small cost the clerk incurs in calculating the location of the data that it wants to retrieve from the remote side. This will be typically on the order of a few tens of microseconds to calculate a hash function, and can be neglected relative to the remaining times.

Notice that in all cases, the pure data transfer scheme does significantly better than the RPC-like scheme. As the amount of data transferred increases, the benefits of separating control and data decrease a little. This is a natural consequence of the fact that the cost of a single control transfer operation is now amortized over a larger data transfer.

Separation of data and control yields better performance for the operations shown in the graph because each operation is logically a simple one involving only data transfer. The costs of context switching and procedure invocation are overheads of the communication primitive. Thus, a specialized data transfer operation is advantageous.

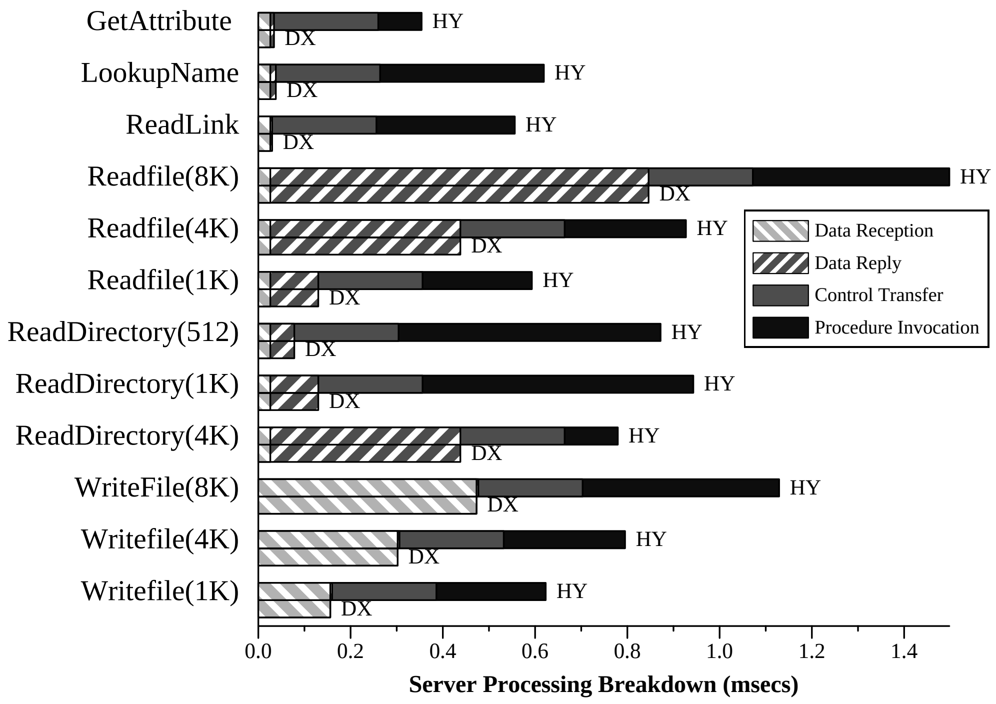


Figure 7.3: Breakdown of Server Activity

Figure 7.3 shows a detailed breakdown of activity on the server CPU. In the pure data transfer scheme, the server CPU is involved only in emulating incoming and outgoing remote memory operations. With a communications co-processor, even this CPU involvement could be eliminated. In contrast, the hybrid model incurs the cost of control transfer and the cost of executing the server's procedure, in addition to the cost of data transfer.

For example, in the LookupName operation, the only server processing for the pure data transfer scheme is the data reception, which is the service-side interrupt-level processing of the remote memory read of the name lookup cache. For the same operation in the hybrid scheme, there are four components: data reception (of the request), control transfer, procedure invocation, and data reply (sending the response), as shown in Figure 7.3. On the average, we see that the pure data transfer scheme imposes less than half the server load imposed by control and data transfer schemes; therefore, the server should be able to accommodate more clients using pure data transfer. Once again, we note that as the amount of data transferred increases, the overhead of control transfer can be amortized more effectively.

7.5 Chapter Summary

We have described a new structure for building distributed systems. Traditionally, distributed systems have been organized around the RPC-based client/server model for a variety of good reasons: a simple programming paradigm, and networks with low bandwidth, high latency, and unpredictable reliability. Changes in technology make it both possible and necessary to re-think the structure of distributed systems. Among the things that have changed are the bandwidth, latency, and reliability of networks.

Our alternative structure is based on the observation that by separating the transfer of control from the transfer of data, we can eliminate one or the other if it is not required. The underpinning for our technique is a set of high-performance network access primitives based on the notion of remote memory. Our technique has three elements to it: (1) server clerks and servers that cache data, (2) an efficient communication primitive that allows quick access to this cached data on remote hosts, and (3) restricting control transfers to occur predominantly within a machine boundary.

Our experiments and measurements with this structure show the promise for improving distributed system performance and scalability through lowered elapsed times and significantly reduced server load. For example, our analysis shows a 50 percent reduction in server load for NFS-style file server operations when using pure data transfer, when compared to a communications model using combined data and control transfer.

The proposed structure is applicable in environments other than distributed systems, e.g., in large-scale dedicated multiprocessors, where the cost of control transfer is high relative to that of data transfer. Traditionally, these multiprocessor systems, e.g., the J-Machine [22], *T [53], etc., have opted for a single primitive that unifies remote transfer of data and control, in contrast to our approach.

It is important to make two final points in passing. First, note that the structure we propose here is one innovative alternative to organizing distributed systems, but not the only one. Second, the proposed structure should be thought of as an application of the network access model, but again, not the only one.

Chapter 8

RELATED WORK

The problem of providing efficient access to networks has been studied in the research community for over a decade using a variety of techniques, both software and hardware. At one stage in the state-of-the-art, software protocols were considered the key factor in performance. This led to several studies that tried to reduce the cost of software processing, e.g., [17], [28], [72] and many others. Reduced protocol costs led to research on the structure of network controllers and other host-architectural effects, e.g., [5], [23], [38], [66] to name a few. The advent of a new generation of ATM networks has, in turn, resulted in many new research efforts, including ours, aimed at higher performance network communication.

The work described in this thesis shares similarities with many previous studies. For example, network access based on a memory abstraction has been previously studied in the literature. Similarly, there have been earlier distributed system designs that are related to the approach suggested in this thesis. The following sections compare our work with previous approaches. For the most part, the discussion below is organized in rough chronological order.

8.1 Spector's Remote References

As far as I am aware, the notion of remote memory access was first suggested by Spector [61]. Spector proposed a taxonomy of remote memory access primitives based on five factors: (1) whether remote references were fault tolerant in the face of network and host failures, (2) whether references returned values or not, (3) whether they were synchronous with respect to the issuing host and process, (4) whether they were flow controlled or not, and (5) whether the remote references were executed directly by a specialized privileged communication process or a regular user process that required additional context switching. The model presented in this thesis draws heavily on Spector's approach but makes significant extensions to support virtual memory, protection, and timeslicing. These issues were largely unaddressed in Spector's work on the Xerox Altos [65], which neither had virtual memory nor address protection among a set of applications.

Spector implemented a small subset of the possible remote access types from his taxonomy. The subset consisted of processor synchronous remote reads, writes, and test-and-set. By implementing the primitives using microcode and software on Altos connected by the experimental Ethernet, Spector demonstrated the feasibility of remote references as a low-level mechanism for data transfer on a workstation cluster. Owing to technology available or anticipated in the late ’70s and early ’80s, Spector’s experiments led him to conclude that realistic multicomputing would not be practical on a cluster of workstations.

In contrast, experience with the model described in this thesis leads us to conclude otherwise. First, it demonstrates that the model can be used as the basis for improving the performance of traditional distributed mechanisms like RPC and for coarse-grained workstation-based multicomputing. Second, it argues that distributed systems can be structured differently by separating the notions of control transfer and data transfer.

8.2 Grapevine

In its use of clerks, our distributed system design has something in common with Grapevine [11]. We have already discussed the relationship between the two systems in Chapter 7, so we only mention the important differences here. Our clerks and servers are more tightly integrated, they are part of the same application, and they trust each other. They do not use RPC for communication, but rely on direct access to each other’s memory.

8.3 VAXclusters

DEC’s VAXclusters system used a set of communication controllers to provide a variety of message operations on a cluster of computers [40]. The controller supported, in hardware, reliable message sends, unreliable datagrams, and block data transfers. The block transfer primitive is similar to a remote write operation, in that the sender specifies where in destination memory the data is to be deposited.

Like our approach, the goal of VAXclusters was to provide a tight integration of machines in a distributed environment. However, unlike our approach, there is a fairly complex protocol for each host to communicate with the communication controller. Further, the controller itself had a complicated implementation. Also, there was no notion of user-level access to the controllers. For example, the major use of the block transfer primitive was to transfer data from the disk server directly to file buffers in kernel memory.

8.4 Mether and Memnet

Mether is another system that provides remote memory accesses on a network of workstations [47]. The primary focus of this system is to provide distributed shared memory (DSM) using the page fault handlers in the VM system. To gain performance, the system has mechanisms to avoid data shipment, leaving the sharing and consistency semantics to the application. In contrast, our model is concerned with the implementation of a general purpose, high-performance communication system suitable for next generation networks and processors. A whole range of DSM protocols can be implemented using our model. Depending on the complexity of the DSM protocols, more or less software needs to be layered on top of our system. We believe that the Mether protocol can be layered quite easily on our model. Thus, though Mether and our design may resemble each other in form, the two are not functionally interchangeable.

The Memnet system, a precursor to Mether, also proposed a memory abstraction to access the network. Memnet implemented a hardware based approach that provided coherent remote page access on a token bus. Like Mether, the focus of this system is on coherent memory abstractions rather than efficient data movement per se.

8.5 DEC AN1

The AN1 network designed at the DEC System Research Center is a high-speed point-to-point network [57]. The network controller for AN1 has hardware support that permits some control over the placement of data at the destination.

In the AN1 network, a single field, called the BQI, in the link-level packet header provides a level of indirection into a table kept in the controller. The table contains a ring of descriptors that specify host memory buffers into which data is transferred. Incoming network packets contain a BQI field that is used by the controller in determining which ring to use. The controller initiates DMA into the next buffer in this ring and hands the buffer to host software.

This mechanism allows some control over data placement, but this is at a much coarser grain than the remote memory model provides. For instance, the sender has no control over where data is placed; only the receiving kernel can do that. Often, the sender knows precisely where the data is to be sent. There are similarities in both approaches, to the extent that both try to provide efficient demultiplexing mechanisms for incoming data.

8.6 Active Messages

Active Messages is a low-level mechanism that has been proposed for communicating between nodes in a dedicated, closely-coupled multicomputer [71]. The key idea in this design is that an incoming message carries with it an upcall address of a handler that integrates the message into the computation stream for the node. Thus, using Active Messages, data and control are very closely-coupled. An Active Message arriving on a node exhibits characteristics similar to that of a network packet arriving on a workstation, with the key difference that the interrupt handler can be different for each message. The general notion of remote memory access (as embodied in our model) is substantially different from the notion of interrupt driven messages that is at the core of Active Messages. In particular, as mentioned in previous chapters, our model explicitly separates the notion of data transfer from control transfer.

In Active Messages, the upcall handler refers, in general, to user code. Dedicated multicomputers like the CM-5 [67], on which Active Messages has been implemented, typically have hardware and network support to ensure that the upcalled user code cannot impact performance of other jobs by misbehaving. Workstation hosts on general purpose networks do not have such mechanisms. Thus, implementing the Active Message model in this environment may require additional functionality, e.g., hardware to allow multiple users to map in the network controller or compiler techniques to ensure that the upcall handlers that execute are well-behaved [35].

8.7 Hamlyn

Wilkes describes the design of a hardware controller for a reliable multicomputer interconnection network [74]. Some of the Hamlyn ideas are similar to the ones described in this thesis. For example, fundamental to the design is the concept of “sender-controlled” data transfer where the sender specifies the location at the receiver where data is to be deposited. Message destinations are kept pinned in memory as long as they are in use. That is, unlike our approach, there is no notion of selectively pinning or unpinning regions of segments. This design choice was made in Hamlyn on the assumption that in a dedicated multicomputing environment the number of senders per node is small enough that each receiver can dedicate memory for each sending node.

The Hamlyn design handles packet notifications using a slightly less general model than ours. In Hamlyn, a message is assumed to be composed of multiple packets which are the fundamental units of data transfer. As packets for a message arrive at the receiver, a preloaded counter associated with the message is decremented; when the counter reaches zero a notification is scheduled. Special hardware is provided to keep these per-message counters.

Protected access to the network is provided by using a hardware structure called a “terminus” that acts as a connection end point on the sender and receiver. The terminus is initialized with protection information by trusted operating system software and mapped directly into the user’s address space. There is a fixed number of termini, which is implementation dependent, that can be used in this manner.

Hamlyn shares many of its goals with our work but is primarily concerned with a dedicated multicomputer environment rather than a workstation cluster. Many of the design choices are therefore different in the two approaches. It is difficult to make a quantitative comparison of the two designs because, at the time of writing, there is no implementation nor any simulation results based on the paper design described in [74].

8.8 SHRIMP

The network interface for the SHRIMP multicomputer being designed at Princeton also proposes direct remote writes to memory [13]. Like Hamlyn and our work, the SHRIMP approach shares a common ancestry with Spector and VAXclusters. However, the main focus of SHRIMP is (1) to provide hardware-supported memory coherence between multicomputer nodes and (2) permit the overlap of communication and computation at the level of individual instructions.

In SHRIMP, buffers in virtual memory are mapped between communicating nodes, which can then directly perform loads to the mapped buffer. In parallel to the processor, the network interface snoops on the memory bus and forwards the request to the remote network interface which updates remote memory. Virtual to physical address mappings are kept in tables in the network interface. When a physical frame backing a virtual buffer page is paged out, the system performs a page shootdown mechanism to other network interfaces that might be remotely storing to the page.

Many of the techniques used in SHRIMP are specific to dedicated multicomputers with specialized hardware. For instance, SHRIMP relies on its ability to selectively change the caching policy of a memory location from write-back to write-through. This is supported on their particular platform but is atypical of hardware platforms. Also, unlike our approach, there are no read operations. Further, the SHRIMP design does not focus on issues of separating control and data transfer in distributed systems.

8.9 Channel Model, V System, Network Objects

The distributed system structure we propose is loosely related to the Channel Model [31], Network Objects [9], and other systems, like V [16], that use RPC for small data and a separate bulk data transport mechanism. Unlike most of these systems, in our model, there is no explicit activity or thread of control at the destination process to handle an incoming stream of data. Also, no specific request is required by the receiver to initiate data receipt.

The V system also proposed the notion of "problem-oriented shared memory," which is related to our idea of using a memory based interface between server clerks and the server. However, the V approach depended on the notion of multicast and remote invocation, neither of which is used in our approach.

Chapter 9

CONCLUDING REMARKS

This dissertation has identified some of the key requirements for high-performance cross-machine communication on next-generation networks. It has tried to argue that, beyond fast processors, networks, and controllers, the view of the network—the "network access model"—can have a big influence on both the performance and the structure of distributed systems. We have demonstrated the design of a network access model that shows good performance and is useful in a variety of situations that arise in current and future uses of a workstation cluster. The key features of the model are that it

- provides efficient and protected user-level access to the network
- eliminates unnecessary data copies
- separates the transfer of control from the transfer of data
- admits a very simple implementation

An important implication of the network access model is that since it decouples the control and data transfer mechanisms, it allows newer organizations of distributed systems. Traditionally, these systems have been organized around the RPC-based client/server model for a variety of good reasons. Changes in technology make it both possible and necessary to re-think the structure of distributed systems. Among the things that have changed are the bandwidth, latency, and reliability of networks.

The significance of new distributed system organizations is likely to increase in the future. As networks speed up, the cost of data transfer is likely to decrease. Relative to the cost of data transfer, the cost of control transfer is not likely to decrease with increased processor speed. Furthermore, we expect remote control transfers to be even less amenable to optimization than local control transfers. Thus, a structure such as the one proposed here that eliminates remote control transfer in favor of local control transfer is likely to be of increasing performance benefit. The organization is also of benefit in distributed memory multiprocessors where the cost of control transfer is much larger than the cost of data transfer because of dedicated network links and interfaces. The approach we suggest is, in fact, quite different from conventional thinking in distributed memory multiprocessor environments, where control and data transfer mechanisms are usually unified.

Technology trends in networks and processors seem to indicate that the distributed systems of the future are likely to resemble the distributed memory multiprocessors of today. Thus, we argue that network interfaces and system structure should be tailored for this environment rather than a loosely-coupled cluster of workstations. A recurring theme in this thesis has been that it is advantageous to have a tight coupling of the network and the processor. This tight coupling between processor and network controller is typically seen in dedicated multiprocessors, but usually not on a workstation. For example, in a multiprocessor, the network controller is tightly tied to the processor and is aware of the cache and memory architecture. The processor manipulates the network interface directly using a few, very simple instructions. The network performance varies depending on whether the network interface is attached to the memory bus or the cache bus. Current experience indicates that locating the network interface closer to the processor gives better performance. We refer to these types of interfaces as *processor-centric*.

In contrast, typical workstation environments contain *I/O centric* network interfaces. These have two characteristics. First, they treat the network as a byte-stream providing primitives to stream data back and forth. Second, they treat the network as a device that requires servicing by the host, and usually notify it by interrupts. They are typically accessed over the I/O bus, contain limited memory and have to be programmed by the host. We refer to them as I/O centric because they are designed with the network in mind, they offer limited support for host accesses, and they are usually oblivious to the host memory architecture. Many aspects of I/O centric network interfaces extract a penalty from the host when it wants to communicate over the network.

A processor-centric implementation of the network access model presented in this thesis appears to be a promising approach to building future network controllers. Using such controllers, next generation processors, and networks, we should be able to build distributed systems that are truly distributed across the network, but whose components are tightly-integrated.

It is undoubtedly the case that the distributed system structure proposed in this thesis is more complex than the traditional one. However, this appears to be a reasonable tradeoff of performance for complexity. This is similar to the tradeoff that is made when uniprocessor programs are migrated to multiprocessors for increased performance. Compiler tools, along the lines of stub generators and protocol compilers, can ease this process. Automatic tools, such as these, can be of enormous benefit in taking specifications of services and generating the server-clerk and server portions of the program. Such tools will allow future implementors of distributed programs the same advantages that stub generators allow implementors of distributed programs today.

In the next decade, we expect to see a tremendous growth in the deployment of regional and national networks. Computers connected to these networks are likely to more concerned with the problems of communication rather than computation. This is to be expected, because we have seen tremendous increases in processor performance while network performance has been more modest. In other words, for efforts like the National Information Infrastructure to be truly successful, research oriented to new communication media, devices, and organization principles is likely to be critical.

The ideas presented in this thesis open up many exciting avenues for the future. The extent to which these ideas enter into the accepted design principles for network controllers, and the design of distributed system services, will ultimately determine the final contribution and impact of the thesis.

Bibliography

[1] Advanced Micro Devices. *Am7990 Local Area Network Controller for Ethernet (LANCE)*, 1986.

[2] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Laboratory for Computer Science, MIT, 1991.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[4] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[5] Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H.T. Kung, Robert D. Sansom, and Peter A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, April 1989.

[6] Brian N. Bershad. *High Performance Cross-Address Space Communication*. Ph.D. thesis, University of Washington, June 1990. Department of Computer Science and Engineering Technical Report 90-06-02.

[7] 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.

[8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[9] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.

[10] Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[11] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

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

[13] Matthias A. Blumrich, Kai Li, Richard Alpert, Zezary Dubnicki, and Edward W. Felten. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–153, May 1994.

[14] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[15] David R. Cheriton and Carey L. Williamson. VMTP as the transport layer for high-performance distributed systems. *IEEE Communications Magazine*, 27(6):37–44, June 1989.

[16] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[17] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–36, June 1989.

[18] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A high throughput transport protocol. In *Proceedings of the 1987 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 353–359, September 1987.

[19] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the 1990 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, September 1990.

[20] Douglas Comer and James Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135, June 1990.

[21] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of the 1994 ACM SIGMETRICS and PERFORMANCE Conference*, pages 150–160, May 1994.

[22] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message-driven processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–196, June 1987.

[23] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[24] Bruce S. Davie. A host-network interface architecture for ATM. In *Proceedings of the 1991 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 307–315, September 1991.

[25] Gary Delp. *The Architecture and Implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*. Ph.D. thesis, University of Delaware, 1988.

[26] Digital Equipment Corporation. *TURBOChannel Hardware Specification*, September 1991.

[27] Digital Equipment Corporation, Workstation Systems Engineering. *PMADD-AA TURBOChannel Ethernet Module Functional Specification, Rev 1.2.*, August 1990.

[28] Willibald A. Doeringer, Doug Dykeman, Matthias Kaiserwerth, Bernd Werner Meister, Harry Rudin, and Robin Williamson. A survey of light-weight transport protocols for high-speed networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[29] Edward W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. Ph.D. thesis, University of Washington, September 1993. Department of Computer Science and Engineering Technical Report 93-09-09.

[30] FORE Systems, 1000 Gamma Drive, Pittsburgh PA 15238. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[31] David K. Gifford and Nathan Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[32] Daniel H. Greene and J. Bryan Lyles. Reliability of adaptation layers. In *Protocols for High-Speed Networks (Proceedings of the IFIP 6.1/6.4 Workshop)*, pages 185–201, 1993.

[33] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[34] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert M. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[35] Wilson C. Hsieh, M. Frans Kaashoek, and William E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Proceedings*

of the Fourth Workshop on Workstation Operating Systems, pages 186–190, October 1993.

[36] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[37] David B. Johnson and Willy Zwaenepoel. The Peregrine high-performance RPC system. *Software – Practice and Experience*, 23(2):201–221, February 1993.

[38] Hemant Kanakia and David R. Cheriton. The VMP network adapter board (NAB): High-performance network communications for multiprocessors. In *Proceedings of the 1988 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 175–187, August 1988.

[39] David Keppel. A portable interface for on-the-fly instruction space modification. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, April 1991.

[40] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[41] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[42] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, December 1993.

[43] Richard J. Littlefield. Characterizing and tuning communication performance on the Touchstone DELTA and the iPSC/860. In *Proceedings of the 1992 Intel User's Group Meeting*, pages 4–7, October 1992.

[44] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high performance internetworking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[45] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[46] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[47] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *Proceedings of the Tenth IEEE Distributed Computing Systems Conference*, 1990.

[48] Steven E. Minzer. Broadband ISDN and Asynchronous Transfer Mode (ATM). *IEEE Communications Magazine*, 27(9):17–24,57, September 1989.

[49] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[50] Ajay Mohindra and Murthy Devarakonda. Distributed token management in a cluster file system. In *Proceedings of the Symposium on Parallel and Distributed Processing*, October 1994.

[51] Bruce Jay Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, May 1981. (Also, Ph.D. thesis, Carnegie-Mellon University, CMU-CS-81-119).

[52] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[53] R. S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[54] J. M. Ortega and R. G. Voight. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(149), 1985.

[55] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[56] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implemention of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, June 1985.

[57] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

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

[59] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA 01803, 1992.

[60] SPEC newsletter benchmark results. Systems Performance Evaluation Cooperative, 1990.

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

[62] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with cache-consistency protocols. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 45–57, December 1989.

[63] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Conference*, pages 191–202, February 1988.

[64] Sun Microsystems Inc., 2550 Garcia Avenune, Mountain View CA 94043. *SBus Specification B.0*, 1990.

[65] Charles P. Thacker, Edward M. McCreight, Butler W. Lampson, Robert F. Sproull, and David R. Boggs. Alto: A personal computer. In *Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, Computer Structures: Principles and Examples*, chapter 33, pages 549–572. McGraw-Hill Book Company, 1982.

[66] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[67] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[68] C. Brendan S. Traw and Jonathan M. Smith. A high-performance host interface for ATM networks. In *Proceedings of the 1991 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 317–325, September 1991.

[69] Lewis W. Tucker and Alan Mainwaring. CMMD: Active Messages on the CM-5. *Parallel Computing*, 20:481–496, November 1994.

[70] R. van Renesse, H. van Staveren, and A. S. Tanenbaum. Performance of the world's fastest distributed operating system. *ACM Operating Systems Review*, 22(4):25–34, October 1988.

[71] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[72] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[73] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65, 1994.

[74] John Wilkes. Hamlyn—an interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett Packard Laboratories, November 1992.

[75] Alec Wolman, Geoff Voelker, and Chandramohan A. Thekkath. Latency analysis of TCP on an ATM network. In *Proceedings of the Winter 1994 USENIX Conference*, pages 167–179, January 1994.

Vita

Chandramohan A. Thekkath received the Bachelor of Technology degree in Electrical Engineering (Electronics) from the Indian Institute of Technology, Madras, in 1982. He received the M.S. degree in Electrical Engineering from the University of California, Santa Barbara, in 1983 and the M.S. degree in Computer Science from Stanford University in 1989. Between 1989 and 1994, he attended the University of Washington, receiving an M.S. in Computer Science in 1992, and his Ph.D in Computer Science in 1994.