Separating Data and Control Transfer in Distributed Operating Systems

Chandramohan A. Thekkath, Henry M. Levy, and
Edward D. Lazowska

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

Technical Report 94-07-04
July 17, 1994

Also appears in the Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

Separating Data and Control Transfer in Distributed Operating Systems

Chandramohan A. Thekkath,[†] Henry M. Levy, and Edward D. Lazowska
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

Advances in processor architecture and technology have resulted in workstations in the 100+ MIPS range. As well, newer local-area networks such as ATM 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.

We believe that these new network and processor technologies will permit tighter coupling of distributed systems at the hardware level, and that distributed systems software should be designed to benefit from that tighter coupling. In this paper, we propose an alternative way of structuring distributed systems that takes advantage of a communication model based on remote network access (reads and writes) to protected memory segments.

A key feature of the new structure, directly supported by the communication model, is the separation of *data transfer* and *control transfer*. This is in contrast to the structure of traditional distributed systems, which are typically organized using message passing or remote procedure call (RPC). In RPC-style systems, data and control are inextricably linked—all RPCs must transfer both data and control, even if the control transfer is unnecessary.

We have implemented our model on DECstation hardware connected by an ATM network. We demonstrate how separating data transfer and control transfer can eliminate unnecessary control transfers and facilitate tighter coupling of the client and server. This has the potential to increase performance and reduce server load, which supports scaling in the face of an increasing number of clients. For example, for a small set of file server operations, our analysis shows a 50% decrease in server load when we switched from a communications mechanism requiring both control transfer and data transfer, to an alternative structure based on pure data transfer.

1 Introduction

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. In addition, newer switch-based local-area networks such as ATM promise significant increase in throughput, much reduced latency, greater scalability, and greatly increased reliability, when compared to Ethernet.

In this paper we consider a novel organization of distributed systems made possible by the new technology and an alternative communication model. It is our contention that new network and processor technologies will permit tighter coupling of distributed systems at the hardware level, and that distributed system structure should change as a result in order to benefit from that tighter coupling.

[†]Author's current address: DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

This work was supported in part by the National Science Foundation (Grants No. 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.

Our communication model consists of a set of primitives to access remote memory. These primitives allow processes on one machine access to a set of remote memory *segments*, which are contiguous pieces of another process' virtual memory. Processes are permitted direct *read*, *write*, and *compare-and-swap* operations at specified offsets within remote segments, which are protected from unauthorized access. Inherent in our model is the separation of data transfer and control transfer, which we believe has the potential to increase performance of distributed applications. We demonstrate the feasibility of the model through a prototype implemented on DECstation workstations connected by a FORE ATM network. We then discuss a new organization of distributed systems built using that model.

Our approach is in contrast to traditional distributed system organizations, which are based on clients and servers that communicate using RPC- or message-based communication. In RPC-style systems, data and control are unified even if one of them is unnecessary. Such RPC-based 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.

2 The Trouble with RPC

RPC is the predominant communication mechanism between the components of contemporary distributed systems. (In this context, we consider RPC and message passing to be essentially identical.) For this reason, an enormous amount of energy has been devoted to increasing its performance.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 and 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 memory and the network.

- 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 and no results, and 7 percent of the overall time for a call with no arguments and a 1440 byte result [17].

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

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

Table 1a: Summary of NFS RPC Activity

	Network Traffic (Mb)		
Activity	**Control**	**Data**	**Control / 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

Table 1b: Breakdown of NFS RPC Traffic

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 1a 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 1a 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 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 1b 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. However, we include file handles, communication identifiers, and marshaling overheads imposed by the RPC system. 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 a new communication model and a new structure for distributed applications.

3 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 discuss the underlying communication model that we have implemented, and describe the structure of distributed applications using this model.

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 resources 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 [1]. 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.

The particular communication model that we use is based on the notion of remote network memory. In this model, processes on one machine can directly read and write memory within the virtual address spaces of processes running on other machines on the network. The remote read/write requests are *data transfer only*; that is, no cooperation is required from the remote process whose address space is being read or written. If control transfer is desired, however, a remote process can be optionally activated through a separate control transfer mechanism. By separating control transfer from data transfer, each can be optimized separately.

3.1 The Communication Model

Our model consists of a set of communication primitives to access remote memory. At an abstract level, this communication 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 implemented by a segment name server. Once exchanged, descriptors can be named by the communicating parties, permitting direct access to data at specified offsets within the remote segments. Data 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.

3.1.1 Memory Instructions

We now present the remote memory model, which is defined as a set of co-processor meta-instructions. (In the next section, we will show how these co-processor instructions can be efficiently emulated without special hardware support. Complete details of the model and its implementation can be found elsewhere [23].) The co-processor contains descriptors that define remote memory segments; each descriptor includes the destination segment size, remote node address, and protection information for a segment. There are three non-privileged meta-instructions supported by the interface: `WRITE`, `READ`, and `CAS` (compare-and-swap).

The 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. In one variant of the write operation, used for small data transfer, the data for the write is taken from a set of message registers shared between the sending processor and co-processor. Another variant of the write, (but not described here) is also available to transfer blocks of data directly from source memory to remote destination memory. *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. The write operations are non-blocking. Further, when a write successfully completes locally, the co-processor only guarantees that the data has been accepted by the network, not that it has been delivered to the destination.

On receiving a write request, the remote co-processor uses the segment descriptor number, the offset, and the size to validate the request. The descriptor identifies a virtual address range within some process. The co-processor reads the address translation tables for that process and writes the data to memory.

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 that the read returned data except by repeatedly checking the destination memory location.

This remote memory model is more efficient than message passing for data transfer, because loads and stores specify the ultimate destination of the data in memory. In contrast, message passing models specify only communication end points, such as sockets, which typically necessitates overhead in demultiplexing and data copying.

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*, *soff*, *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 will contain the result of the compare-and-swap, which is either success or failure.

Many characteristics of a workstation cluster make it necessary to provide functionality beyond simple reads and writes. These are related to sharing, protection, independent time-slicing among nodes, and other factors. Thus, in addition to simple memory instructions, our model and implementation explicitly support mechanisms to accommodate these special needs. 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) control transfer. Most of these details do not concern us here, however we describe the control transfer mechanism below.

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 process 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.

Our model does not impose or support a particular control transfer mechanism. The transfer of data is not tied to the execution of a particular thread or a procedure. The model we propose is flexible yet simple and is amenable to very efficient implementations.

Our model has much in common with Spector's remote references [19]. Like his approach, we have chosen simple primitives that allow an efficient implementation on contemporary workstation clusters. We have extended the remote access primitives to incorporate virtual memory, protected access, and time-slicing on workstations. These issues were largely unaddressed by Spector's work on the Alto [22], which had neither virtual memory nor enforced address protection among a set of applications. Spector distinguished between *primary* operations that were performed by a special communication process, and *secondary* operations that were performed by a regular process. Secondary operations involved context switch overheads and packet demultiplexing and were meant to implement long running remote operations. Primary operations were meant to implement simple operations like remote fetch and store. This distinction is loosely related to our notion of separating data and control.

3.1.2 Implementation Overview and Performance

We have implemented the remote memory model using a software layer on top of an existing ATM host-network interface. We use FORE TCA-100 host-network interfaces to connect our DECstation 5000s to a 140 Mb/s ATM network.The interface is located on the 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. Processor accesses to these FIFOs are performed a word at a time.

The bulk of the communication model is implemented in system software running inside the kernel. The

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

Table 2: Performance Summary of Remote Memory Operations

performance critical meta-instructions (e.g., `READ`, `WRITE`, and `CAS`) are implemented as MIPS machine instructions, using unused opcodes from the R3000 instruction set. Thus, these meta-instructions can be directly executed by user programs; meta-instruction execution causes a rapid trap to a carefully tuned assembly routine, which emulates that instruction in the kernel. The implementation guarantees that a single-word local access (read/write) is atomic with respect to a remote access (read/write) involving that word. All the system software and emulated instructions are integrated into an otherwise standard DEC Ultrix kernel. Protection is provided by the emulation code, which checks the validity of all remote accesses using in-kernel tables containing address translation entries for each segment. In the case of data-only transfer, remote requests do not require the receiving process to take any action. The co-processor emulation code in the kernel does all the necessary processing.

Since our model is implemented by software emulation in the Ultrix kernel, we have integrated control transfer using file descriptors. Associated with each segment is a file descriptor that the user can access. Normally, a descriptor access will block the process. However, whenever an incoming operation specifies a control transfer operation, the file descriptor becomes ready for reading with a small amount of control information. Using the standard "`select`", "`read`", "`fcntl`", and "`signal`" system calls, a process has some degree of flexibility in receiving notifications. When a remote request requiring user notification arrives on a node, the system invokes a user-specified signal handler procedure.

Table 2 summarizes the performance of our implementation. The latency represents the elapsed time for performing single-cell accesses. Each single-cell read and write operation moves 10 4-byte words. 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.

As Table 2 shows, we achieve a latency of only 30 μs for a remote write operation containing one ATM cell (40 data bytes). For comparison, a processor-local write of that size is only 15 times faster on the same hardware. 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.

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 the performance that the raw controller hardware is capable of.

3.2 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 communication model described in the previous section 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 the previous section 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 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 [2, 13]. 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.

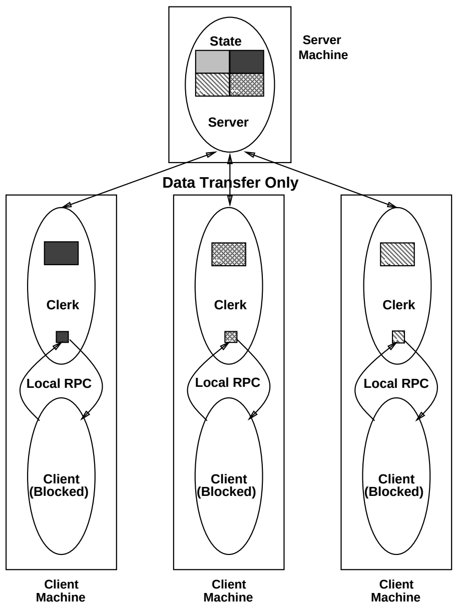

Figure 1: Structuring a Distributed Service Without RPC

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 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 [5]. 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.

3.3 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 4.1 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.

3.4 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, strict synchronization of the data is not required 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 4.1.

A third option is to use the synchronization provided by the CAS operation supported by the communication 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 communication 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 when necessary. 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.

3.5 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 communication 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 [4, 21]. 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 our communication 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 [18] 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 communication model and the structure that we propose can be used in a secure fashion.

3.6 Heterogeneity

It might appear that the proposed structure and communication 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 communication 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.

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.

3.7 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 [15], SNFS [20], and AFS [11], 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.

4 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.

4.1 Design of the Simple Name Server

Recall that our communication primitives rely on 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 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, segments 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.

4.2 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. 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.

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

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

Table 3: Name Server Performance

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.

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 user-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.

4.3 Performance

Table 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 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 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, although more expensive option is to not spin wait the importer, but instead have the exporter use a remote write with notification.

5 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 often an unneeded cost imposed by the use of RPC between clients and servers. Recall from Section 2 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.

5.1 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. (Even NFS has a cache-consistency policy, albeit a weak one.)

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 [14] 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 as shown below. This organization allows the client-side server clerk to probe server data structures. This 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 1a.

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 three 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.

5.2 Performance

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: the total latency seen by a client and 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. 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 5.1) 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 1a.

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.

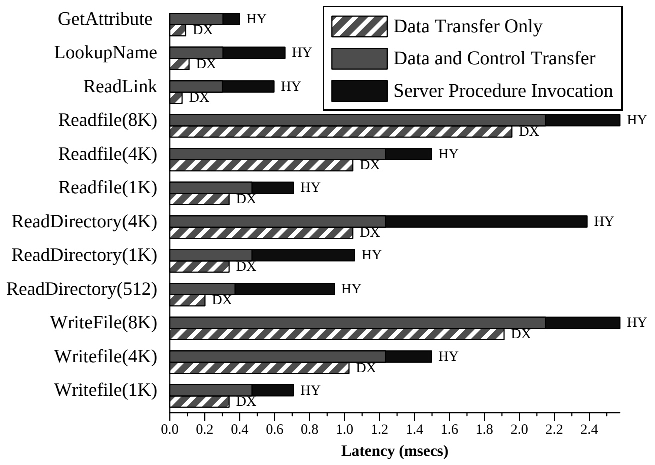

Figure 2: Request Processing Latency Seen by Client

Figure 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 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

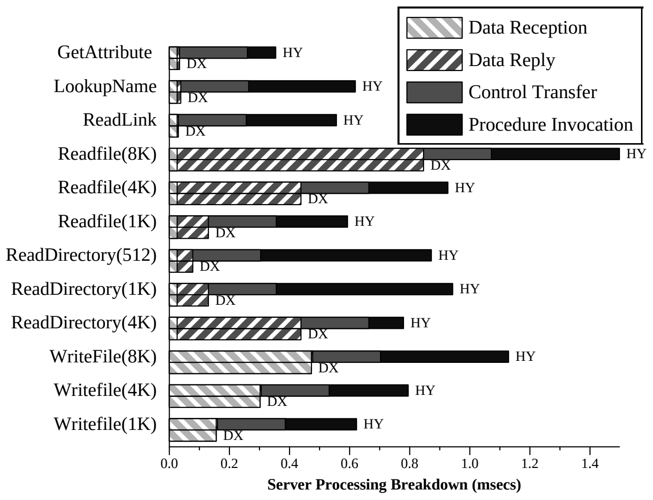

Figure 3: Breakdown of Server Activity

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 data reception and reply, which is the service-side 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 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.

6 Related Work

Section 3.1 has already described our relationship to previous work like Grapevine [5] and Spector's remote reference primitives [19]. This section discusses other related systems.

Shared Virtual Memory (SVM) systems like Ivy [12] are related to our approach. In fact, we can implement our system organization over an SVM system. Further, with SVM systems, the unit of sharing and data transfer is usually a page, which in modern processors can be upwards of 4K bytes. This large size might lead to false sharing between clerks resulting in suboptimal performance. Finally, most SVM implementations require non-trivial processing and control transfer at the machine that faults the page in, which is contrary to our approach. Specialized SVM systems like MemNet [9] have been implemented in hardware to gain performance. However, by requiring all shared memory to be coherent, these schemes rely on complex network interfaces and sometimes on broadcast primitives from the network. In contrast, our system organization relies on the existence of very simple communication primitives and requires no coherency guarantees.

Our system is also related to the Channel Model [10], Network Objects [3], and other systems, like V [7], 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.

Active Messages is a low-level mechanism that has been proposed for communicating between nodes in a dedicated, closely-coupled multicomputer [24]. 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. 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 sections, our model explicitly separates the notion of data transfer from control transfer.

Network interfaces for multicomputers like SHRIMP [6] and Hamlyn [25] share a common ancestry with us to Spector's work. Both of these interfaces use remote writes as the basic mechanism for data transfer and use segment or page descriptor based schemes to ensure protection. There are many differences as well, e.g., SHRIMP focuses on providing memory coherence. Neither system is primarily concerned with structuring distributed systems by separating control and data transfer.

7 Concluding Remarks

This paper has described a new structure for building distributed systems. Traditionally, these systems have been organized around the RPC-based client/server model for a variety of good reasons: a simple programming paradigm, networks have 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 and latency of networks, and increased reliability.

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 systems performance and scalability through lowered elapsed times and significantly reduced server load. For example, our measurements show a 50 percent reduction in server load for NFS-style file server operations using pure data transfer, as 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 [8], *T [16], etc., have opted for a single primitive that unifies remote transfer of data and control, in contrast to our approach.

Acknowledgments

The authors gratefully acknowledge Alex Klaiber's help in improving the graphical presentation of the paper. We would also like to acknowledge the referees for their comments and help in improving the clarity of the paper. Thanks are also due to FORE Systems Inc. for the loan of ATM networking equipment.

References

[1] T. E. Anderson et al. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, Nov. 1993.

[2] B. N. Bershad et al. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, Feb. 1990.

[3] A. Birrell et al. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Dec. 1993.

[4] A. D. Birrell. Secure communication using remote procedure calls. *ACM Trans. Comput. Syst.*, 3(1):1–14, Feb. 1985.

[5] A. D. Birrell et al. Grapevine: An exercise in distributed computing. *Commun. ACM*, 25(4):260–274, April 1982.

[6] M. A. Blumrich et al. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–153, May 1994.

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

[8] W. J. Dally et al. Architecture of a message-driven processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–196, June 1987.

[9] G. Delp. *The Architecture and Implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*. PhD thesis, University of Delaware, 1988.

[10] D. K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Trans. Comput. Syst.*, 6(3):258–283, Aug. 1988.

[11] J. H. Howard et al. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

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

[14] A. Mohindra and M. Devarakonda. Distributed token management in a cluster file system. To appear in *Proceedings of the Symposium on Parallel and Distributed Processing*, Oct. 1994.

[15] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, Feb. 1988.

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

[17] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Trans. Comput. Syst.*, 8(1):1–17, Feb. 1990.

[18] M. D. Schroeder et al. 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, Oct. 1991.

[19] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, April 1982.

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

[21] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Conference*, pages 191–202, Feb. 1988.

[22] C. P. Thacker et al. 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.

[23] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Efficient support for multicomputing on ATM networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, Apr. 1993.

[24] T. von Eicken et al. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

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