1990

Implementation and Performance of a Communication Facility for Distributed Transaction Processing

Enrique Mafla

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
90-1046

IMPLEMENTATION AND PERFORMANCE OF A COMMUNICATION FACILITY FOR DISTRIBUTED TRANSACTION PROCESSING

Enrique Mafla
Bharat Bhargava

CSD-TR-1046
November 1990

Implementation and Performance of a Communication Facility for Distributed Transaction Processing*

Enrique Mafla
Bharat Bhargava
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

We identify the problems in general purpose interprocess communication mechanisms available for the Raid distributed database transaction processing system by conducting a series of experiments. These mechanism are CPU intensive, optimized only for remote communication and do not support multicasting. We develop a transaction-oriented communication facility to address these problems. Its performance is limited only by the network device driver and the system call mechanism overheads. Sending a 100-byte message (monocast or multicast) takes 650 microseconds, which includes the overheads. This is approximately 30% of the cost of the corresponding Unix communication facility. This communication facility demonstrates the feasibility of address spaces for structuring complex distributed transaction processing systems. The new communication facility employs shared-memory ports, a simple naming scheme, and a transaction-oriented multicasting mechanism. Local and remote communication is through ports. Ports can be accessed directly by the kernel and by user-level processes. The naming scheme used for the application and network levels avoids the use of name-resolution protocols by directly mappings the application-level name space to the network name space. Physical multicasting is used and the need for special protocols to agree on a group address is avoided. Each transaction defines a multicasting group consisting of the set of sites involved. A group's multicasting address is a function of the corresponding transaction identifier and can be independently determined by each member of the group. The new communication facility reduces kernel overhead during transaction processing in Raid by up to 70%.

*This research is supported by NASA and AIRMICS under grant number NAG-1-676, NSF grant IRI-8821398, and AT&T.

1 Introduction

Transaction processing systems are large and complex systems. Address spaces can be a useful structuring device for such systems [Ber90]. Under that structuring paradigm, each of the logical components of the system is implemented in a separate address space. The separation of the logical components of the system is enforced by hardware. Many operating systems use address spaces to isolate their functional components [YTR+87, RR81, DJA88, Ben87]. Several distributed transaction processing systems have also been developed under this paradigm [LCJS87, BR89b, Duc89].

Expensive interprocess communication facilities results in systems with either poor structure or poor performance [vRvST88, BALL90]. The *small-kernel* or *backplane* paradigm for structuring distributed systems offers several advantages [Ber90, DJA88]. The performance of such system designs demands efficient interprocess communication services [Che84]. In many distributed systems, the interprocess communication overhead is such that designers are forced to compromise the structuring of the system on behalf of performance [Ber90, RR81, LMKQ89].

Interprocess communication have been a topic of study in many experimental systems. Lightweight RPC is a high performance cross-address space communication method, which exploits the fact that most communication is local rather than remote [BALL90]. User-level RPC frees the kernel from communication processing [Ber90]. In some systems, the integration of virtual memory, interprocess communication, and file systems have been used to improve their performance [RR81, YTR+87]. This is also being investigated to improve the communication support for distributed transaction processing systems. In order to reduce communication overhead, Argus uses threads and runs on a modified Unix[1] kernel [LCJS87]. In Camelot, threads have been used to improve its throughput [Duc89].

We study this problem in the context of the Raid system [BR89b]. Raid is a robust and adaptable distributed database system for transaction processing and has been developed on Sun[2] workstations under the Unix operating system. Raid is structured as a server-based system to provide the infrastructure for adaptability [BR89a]. Each major functional component of Raid is implemented as a server, which is a process interacting with other processes only through a high-level communication subsystem, which has been presented in [BMR90]. Currently, there are six major subsystems in Raid: User Interface (UI), Action Driver (AD), Access Manager (AM), Atomicity Controller (AC), Concurrency Controller (CC), and Replication Controller (RC).

[1] Unix is a trademark of AT&T Bell Laboratories.

[2] Sun is a trademark of Sun Microsystems, Inc.

1.1 Objectives of our Research

Our goal is to develop efficient communication support for distributed transaction processing systems in conventional architectures. By conventional architectures, we mean virtual-memory, single-processor machines with no special hardware support for interprocess communication[3]. Our efficiency measure is the ratio between the CPU time spent on communication and the CPU time spent processing transaction management algorithms. This measure of efficiency is especially appropriate for local area networks, where network delay is insignificant compared to message processing times.

Address spaces can provide a natural platform for the support of concurrency, reliability, and adaptability. The complete system (including kernel, operating system services, and applications) is decomposed into smaller and simpler modules. The modules are self-contained and interact with each other via well-defined interfaces. First, concurrency benefits because processes are the schedulable units of the system. An input-output interaction will not block the whole system, but only the module that handles that interaction (e.g. the disk manager, or the communication manager). The other modules of the system can still run concurrently. Second, reliability increases because of the hardware-enforced failure isolation of the logical modules of the system. Finally, it is easier to implement short and long term adaptability [BR89a].

In this paper, we identify the mechanisms for efficient communication for transaction processing in the Raid distributed database system. We plan to measure message traffic, message processing costs, and communication overhead in Raid [BR89b]. Next, we design and implement a transaction processing oriented communication mechanism that addresses these issues. Finally, we compare the performance of the transaction processing system running on both a conventional and the new communication subsystems. We contrast the two options based on the ratio: communication-time/processing-time. Communication time is the CPU time overhead introduced by communication. Processing time is the CPU time dedicated to process transaction processing functions: concurrency control, atomicity control, replication control, etc.

1.2 Experiences from our Experimental Research

We discuss our experiences with interprocess communication support for distributed transaction processing in two steps. In the first step, we get a better understanding about the problems in present communication facilities. In the second step, we rectify the problems and implement a new communication facility.

1. The first lesson is an understanding of the problems in conventional communication facilities in supporting distributed transaction processing systems.

[3]Some mainframe computers have hardware assistance for IPC. More than one address space can be accessed at the same time.

- General-purpose communication facilities are top-heavy. To support generality, they use complex and expensive communication abstractions and mechanisms. For instance, in the socket-based interprocess communication model of Unix, the socket abstraction and the *mbuf*[4] mechanism are the platform for the implementation of a variety of communication protocols. However, they are responsible for most of the communication overhead (see section 3.1). In addition, messages have to go through all layers of the communication subsystem even if some of them are irrelevant to the processing of specific messages. For example, the internetwork protocol layer is needed only by messages that cross network boundaries.
- Interprocess communication services are designed with the remote case in mind. The local case is handled as a particular instance of the remote case. This results in inefficient local communication services. There are several efficient alternatives for local communication. However, those alternatives are difficult to integrate with the remote communication facility (see subsection 3.2). The inefficiency of local communication is a serious problem for distributed transaction processing systems. In many distributed systems, communication is predominantly local [BALL90].
- Name resolution can become an expensive and complicated process. In general, we can have three different name spaces: application name space, interprocess communication name space, and network name space. If the name spaces are not related in a simple way, name resolution protocols have to be used for mapping names from one space into another. For instance, the Raid system uses a special protocol to map Raid names into interprocess communication addresses (UDP/IP addresses). These addresses have to be mapped into network addresses (e.g. Ethernet addresses) via a second address resolution protocol.
- Some operating systems do not provide enough communication support for distributed transaction processing. The transaction processing system implementor has to supply those services. Transaction processing systems need special multicasting support. In those systems, multicasting groups are short-lived. The groups are meaningful only for the duration of a transaction, e.g. in commitment protocols or even only for the duration of a single transaction operation, e.g. in quorum-based protocols (see section 3.3). It is desirable to define high-level interfaces between the modules of complex system [BFH+90, RR81]. For communication, modules use typed messages rather than simple buffers of bytes supported by the operating system. To send a message, it has to be marshaled into kernel buffers. The receiving side has to perform the inverse operation. Those are very costly operations (see figure 4).

[4] *Mbufs* are the units of memory allocation used in the Unix communication subsystem.

- Distributed transaction processing are communication intensive. In a well-decomposed system, most of the communication is local rather than remote. The operating system kernel becomes the bottleneck because of both the high message traffic and the high cost to process messages (see figure 4).

2. The second experience is obtained through the design and implementation of a transaction-oriented communication facility for local area networks. The communication facility has been achieved with the following features:

   - Communication takes place between ports. Ports can be accessed directly by the kernel and by user-level processes, which avoids copying overheads.
   - Multicasting is optimized for transaction processing. It is both CPU and network efficient.
   - There is a simple name mapping between the application level and the network. It avoids the use of special address resolution protocols.
   - The kernel provides lightweight communication services for both local and remote messages.
   - The use of this communication facility for transaction processing results in savings of kernel time of up to 70%.

In the next section, we identify other projects on this problem. In section 3, we study the problems with conventional interprocess communications facilities to support distributed transaction processing. Based on those studies, we designed and implemented a new communication mechanism to support distributed transaction processing systems. Section 4 describes this new communication mechanism. Section 4.3 presents performance figures for the new communication mechanism. In that section, we also test the impact of the new communication subsystem on the performance of the Raid distributed database system. We end this paper with conclusions and suggestions for further work in this area.

2 Observations and Paradigms in Related Research Efforts

Many communication facilities were originally developed for wide area networks where network delays are considerable. The main applications of computer communication were limited to remote terminal access and file transfer [Che86]. For those applications, point-to-point virtual circuits implemented on top of expensive layers of communication protocols provided an adequate support. Several special-purpose communication facilities have been proposed to support distributed systems in local area networks. In this section, we describe some of

the most relevant communication observations and paradigms that have been developed to support distributed systems.

We classify the work in this area in five groups: interprocess communication support for distributed systems, local interprocess communication, general purpose communication protocols, communication in local area networks, and communication support for the implementation of distributed transaction processing algorithms.

2.1 IPC in Distributed Systems

Accent is an example of a communication oriented system [RR81]. The kernel provides support for interprocess communication, virtual memory management, and process management. Operating system services are implemented in layers of processes built on top of the kernel. To improve performance, Accent intergrates virtual memory support, file management, and interprocess communication. Communication takes place between ports. Ports are protected by the kernel using capabilities. Messages are typed and can include pointers. Software interrupts can be used to receive messages asynchronously. Despite all the emphasis on performance, some efficiency-critical applications require the merging of operating system services (e.g. the network server) with the kernel [RR81].

Mach is a successor of the Accent distributed operating system [YTR+87]. Its interprocess communication subsystem takes advantage of the system's virtual memory mechanism and scheduling policy. Interprocess communication uses the page mapping mechanism of the virtual memory system to optimize bulk data transfer between local processes. Handoff scheduling is used to reduce context switching overhead [JC89]. Handoff scheduling addresses the limitations of traditional time-sharing schedulers for parallel and concurrent programs. Message-intensive applications running on Mach still have performance problems. The Camelot distributed transaction system is implemented on top of Mach [STP+87]. Experiments detected that the load on the operating system is considerable higher than the load on the any part of Camelot [Duc89]. To quote the authors of those experiments: "this is an unavoidable consequence of implementing the transaction manager as a service reachable only via interprocess communication" [Duc89].

The Versatile Message Transaction Protocol (VMTP) is a transport level protocol intended to support the intra-system model of distributed processing [Che86]. It is optimized for page-level file access, remote procedure calls, real-time datagrams, and multicasting. These communication activities dominate in a distributed processing environment. In order to support conversations at the user level, VMTP does not implement virtual circuits. Instead, it provides two facilities, *stable addressing* and *message transactions*, which can be used to implement conversations at higher levels. A stable address can be used in multiple message transactions, as long as it remains valid. A message transaction is a reliable request-response interaction between addressable network entities (ports, processes, procedure invocations). Multicasting, datagrams, and forwarding services are provided as variants

of the message transaction mechanism.

DUNE's *service request* model extends the remote procedure call model [PA89]. In remote procedure calls, the transmission of data and results across process address spaces is limited to the beginning and end of the remote procedure call. In the service request model, client and server can interact in the middle of the call. This allows a dynamic binding of the amount, source, and destination of data involved in the call.

The performance of server-based transaction processing systems like Raid is also affected by the efficiency of the underlying interprocess communication subsystem. Server-based transaction processing systems are communication-intensive [Maf90]. The integration of the file system, virtual memory mechanism, and interprocess communication facility can have a positive effect on the performance of database transaction systems, where considerable amounts of data have to be passed among servers. The control flow in transaction processing can be determined in advance since messages follow well defined paths between servers. Because of this property, a transaction processing system can take advantage of handoff scheduling. The intra-system model of communication introduced by VMTP can reduce the complexity of the design and implementation in transaction processing systems. Specialized multicasting support is needed for replication, commitment, and recovery control in transaction processing Sophisticated RPC services like those supported in Dune can be useful for the implementation of object in transaction processing systems.

2.2 Local IPC

Most of the interprocess communication activity in a distributed system is local [Ber90]. Despite this fact, conventional communication mechanisms are optimized for the remote case. To avoid the expensive cost of cross-address space communication, implementors are forced to place several functional modules into the same address space. This will certainly increase the performance of the system, but at the same time it will decrease the system reliability, modularity, and extensibility. To get high performance without affecting those properties, two new communication facilities are introduced in [Ber90]. The first is called Lightweight Remote Procedure Call. It takes advantage of the control transfer and communication model of capability systems and the address space based protection model of traditional interprocess communication facilities. The second cross address space communication facility, called User-Level Remote Procedure Call, eliminates the role of the kernel as an interprocess communication intermediary. Communication and thread management code is included in each user address space.

In [Che84], the advantages of interprocess communication for structuring operating systems have been discussed. In order to improve performance, the processor's registers are used as a communication channel between processes. This approach avoid copying and context switching overhead. Copying is avoided because the registers are physically shared. Context switching overhead is reduced because there are less registers that need to be saved and

restored. The practical use of this mechanism is limited by the maximum message length allowed. In addition, this interprocess communication method needs special scheduling policies to produce positive results (the receiving process has to run immediately after the sending process relinquishes the processor).

Shared memory and user-level synchronization have been used in a Firefly multiprocessor in order to increase interprocessor communication performance [SB90]. Processes share a pool of messages. Message passing does not involve copying among address spaces. Only pointers are passed among processes. Access to messages is synchronized with the use of single locks, which are mapped to each process' address space.

Like other distributed systems, Raid has need for intensive local communication activity [Ber90, Maf90]. For a transaction with six-operation in a five-site distributed database system, 90% of the communication activity has been found to be local [Maf90]. The user-level remote procedure call and the communication facilities implemented in the Firefly multiprocessor are attractive for supporting efficient local interprocess communication in Raid. Those facilities are based on shared-memory, which can simplify and speed up the transport of high-level, typed messages with pointers among servers.

2.3 Communication Protocols

Virtual protocols and *layered protocols* are used in the *x*-kernel to implement general-purpose yet efficient remote procedure call protocols [HPAO89]. RPC mechanisms are implemented on top of virtual circuits. Those are demultiplexers that route the messages to appropriate lower-level protocols. For example, in an Internet environment, the virtual protocol will bypass the IP protocol for messages originating and ending in the same network. Layered protocols favor the reuse of code through the mapping of a protocol's functional layers into self-contained software modules. Communication protocols are implemented assembling those modules in a given order. This approach is also used in System V Unix streams [Rit84]. Streams are full-duplex connections between a process and a device or another process [PR90]. Initially, a stream consist of two modules, each facing one end of the communication path. Streams can be customized by adding processing modules between those end points. Streams can be seen as the kernel counterparts of Shell pipelines, except that data flow is bidirectional.

Most distributed systems have been developed for local area networks. Interprocess communication in wide area networks and especially in internetworks demands more sophisticated and expensive communication protocols. However, not all messages make use of the full functionality of those protocols. The abstractions discussed above allow the specialization of communication services, which can result in savings of processing time for certain messages. Control flow for distributed transaction processing systems in wide area networks and internetworks can be adapted in order to minimize the traffic of costly long-distance or inter-network messages. Techniques like batching and piggybacking can also be used to

increase efficiency in these environments.

2.4 Communication in Local Area Networks

In [Svo86], the need for specialized communication protocols for local area networks has been argued. The overhead of standard protocols cancels the high communication speed offered by modern local area network technology. Several experiments have shown that communication in local area networks is CPU intensive [BMR87, LZCZ86]. Application-oriented protocols provide opportunities for further optimization. Efficient streamlined protocols for high-speed bulk-data transfer have been implemented and used in local area networks [CLZ87, Zwa85]. Reliable multicasting schemes for local area networks make use of their technology to optimize resource utilization and communication delay [KTHB89, VM90].

2.5 Communication Support for Distributed Transaction Processing

The functions that a communication subsystem should provide in order to support a distributed transaction processing systems have been identified in the Camelot project [Spe86]. RPC-based session services have been proposed to support the interaction among data servers and applications. Besides the synchronous RPC with *at-most-once* semantics, other forms of RPC such as asynchronous RPC and multicasting RPC are useful. Highly specialized datagram-based communication facilities can be used to satisfy the performance demands of data servers and applications. The communication subsystem can use its knowledge about the nature of the transaction system to improve its services. For example, it can record the addresses of transaction participant sites to assist the transaction manager at commitment time.

Specialized communication facilities have been used to improve the design and implementation of distributed database management systems. Chang [Cha84] introduced a two-phase non-blocking commitment protocol using atomic broadcast. The support of atomic broadcasting and failure detection within the communication subsystem simplifies database protocols and optimizes the use of the network broadcasting capabilities. Birman [BJ87] uses a family of reliable multicasting protocols to support the concept of *fault-tolerant process groups* in a distributed environment. Agreement protocols are used to guarantee the consistency of distributed data. These protocols demand expensive interchanges of messages among data server processes. The broadcasting capabilities of local area networks can be used to avoid those message exchanges [MSM89, KTHB89].

3 Problems with Present Communication Schemes

Despite the advances made, few distributed operating systems are currently in commercial use [vRvST88]. This is due to an unacceptable communication overhead in these systems [Duc89, Che84]. The purpose of this section is to identify the problems with present interprocess communication facilities to support distributed transaction systems structured around address spaces. We conduct this research by using Raid as an example of such a system. Raid servers represent functions (such as concurrency, atomicity, replication) that are widely recognized as the logical components of a distributed transaction processing system. Raid is implemented on Unix, which is a monolithic operating system. In Raid, each logical component is implemented in a separate address space. Servers can migrate to different computing nodes to increase reliability, load balancing, adaptability, and availability.

We conducted a series of experiments on the performance of the Raid communication subsystem. The results from these experiments helped us understand the weaknesses of the current communication support for transaction processing, and guided us to the development of new communication services that support transaction processing more effectively. Our Raid laboratory is equipped with five Sun 3/50s and four SPARCstations connected to a 10 Mbps Ethernet. The Raid Ethernet is shared with other departmental machines and has connections to other local area networks and to the Internet. All Raid machines have local disks and are also served by departmental file servers. They run the SunOS 4.0 operating system. For our experiments, we used only the Sun 3/50s. One of the workstations is configured with a special microsecond resolution clock, which is used to measure elapsed times[5].

In this section, we briefly describe our experimental work in the following areas: general purpose interprocess communication facilities, local communication, multicasting, and impact of interprocess communication on distributed transaction processing performance. Details on the experimental setup, procedures, and analysis can be found in [Maf90].

3.1 Experiment I: Socket-based Interprocess Communication

We studied the performance of conventional implementations of interprocess communication services. Our goal is to identify the overhead of layered implementations of communication protocols as well as the overhead of sophisticated interprocess communication abstractions. Layered implementations and complex interprocess communication abstractions are used to support generality. Unfortunately, they compromise the efficiency of the communication facilities. Particular applications running on special environments, (e.g, transaction processing on local area networks) do not make use of that generality. However, those application suffer

[5]This timer board was developed by Peter Danzig and Steve Melvin. It uses the timer chip AM9513A from Advanced Micro Devices, Inc. The timer has a resolution of up to four ticks per microsecond. The overhead to read a timestamp is approximately 20 μs.

the large overhead of the communication services anyway [Svo86]. We use the 4.3BSD Unix socket model of interprocess communication for our experiments.

We measured the overhead introduced by the layers of the socket-based interprocess communication model for datagram communication (UDP). These layers include the system call mechanism, the socket abstraction, communication protocols (UDP, IP, and Ethernet), and interrupt processing. For the send operation, we also measured the startup overhead of the network board. The receive side requires significant context switch activity. We added this overhead as a separate item for the receive operation.

Figure 1 shows the contribution of each layer to the total time of the send operation. The

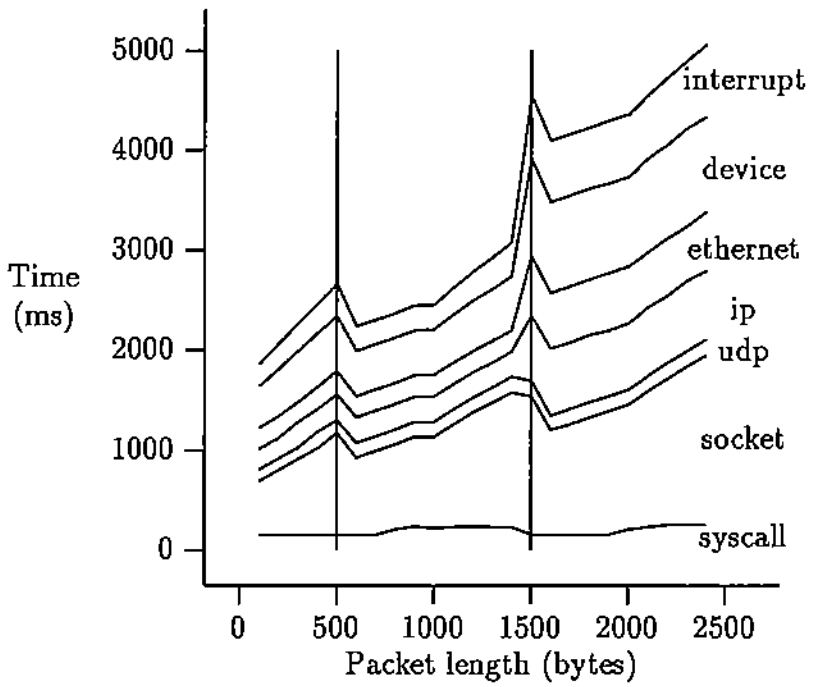

Figure 1: Timing for UDP send (in μs)

receive side presents a similar picture [BMR90]. Lengths are those of user-level messages. This detailed picture reveals the basic problems of the socket-based interprocess communication facility. The most expensive layer is the socket abstraction. It includes copying the message between user and kernel spaces. Starting the physical device requires approximately 20% of the total send time. Protocol processing itself is relatively cheap. Processing of the three communication protocols incidental to IPC (UDP, IP, Ethernet), represents less than 30% of the total time (except for packets that require assembling on the receive side). On the sending machine, each fragment requires starting the device and processing a network interrupt. On the receiving machine, each fragment requires an interrupt processing and the scheduling and execution of the software interrupt routine.

Figure 1 shows the effect of the IPC memory management subsystem. The IPC memory unit is called an *mbuf* [LMKQ89]. Mbufs can contain up to 112 bytes of data by themselves or up to 1024 bytes on a separate page. User messages are copied into mbufs in kernel space. If the message is larger than 560 bytes, 1024-byte mbufs are allocated. Otherwise, 112 byte mbufs are used. For example, a message 560 bytes long uses five 112 byte mbufs, while a 561 byte message uses a single 1024 byte mbuf. Most of the overhead of passing a packet from layer to layer is proportional to the number of mbufs used, rather than to the size of the packet. Because of this memory management policy, sending a 1K message is cheaper than sending a 512 byte message. The vertical lines in the figure show the changes to 1024 byte mbufs. In the figure, the changes are shown for 500 and 1500 bytes. This is because we changed the packet length by intervals of 100 bytes.

3.2 Experiment II: Local Communication

Each Raid server runs in its own address space, communicating with other servers through kernel-based mechanisms. However, systems structured this way will have a significant local cross-address space communication activity. In a five-site Raid system, roughly 90% of the communication is local to one machine. To achieve good performance, we need efficient local interprocess communication support. Traditional interprocess communication mechanisms are optimized for the remote case [BALL90]. In this experiment, we investigate efficient alternatives for local interprocess communication. We have seen in the previous subsection that socket-based communication is not only expensive but also biased against the local case.

The multi-server per process approach has been used in Raid to avoid the high costs of local IPC and interprocess context switch [KLB89]. Several servers are merged into the same process. Under this approach, inter-server communication takes the form of simple data movement within the same address space. Furthermore, the number of context switches needed during transaction processing is drastically reduced. The use of this paradigm in the implementation of the Raid model has led to improvements in transaction execution times of up to 40 percent [KLB89].

We measured the round-trip performance of several local interprocess communication facilities available on SunOS. We investigated the following communication mechanisms: two message queues, one-message queue, named pipes, shared memory, and UDP sockets in both the Internet and Unix domains.

The results of the measurements conducted on these methods are displayed in figure 2. The figure demonstrates the benefits of special-purpose methods for local IPC. All methods have a constant overhead (null message times) and a variable overhead proportional to the message length. The picture for UDP is more complex for reasons explained in section 3.1.

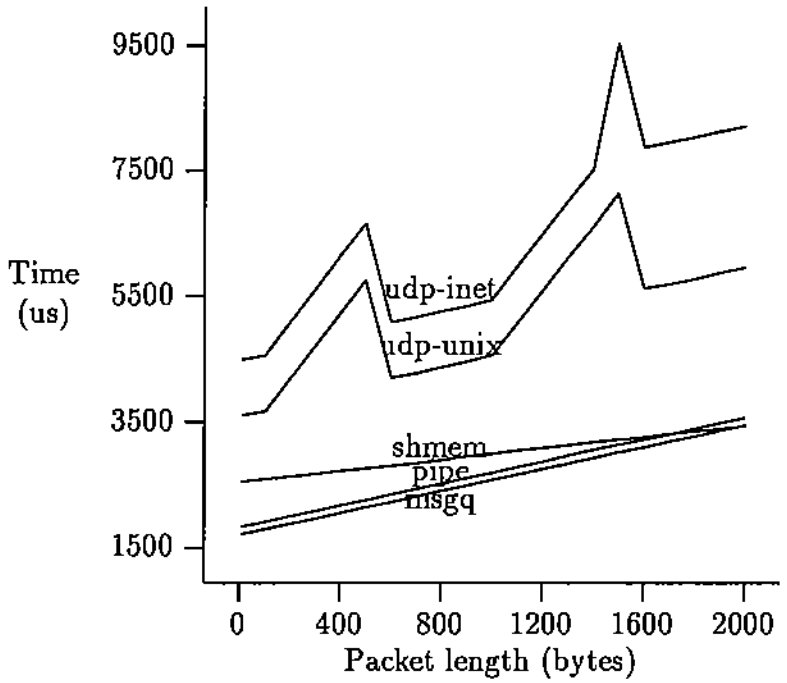

Figure 2: Round trip times for local IPC methods

3.3 Experiment III: Multicast Communication

Replication and commitment protocols require multicast messages. Many operating systems do not provide any multicasting facility. Applications have to simulate multicasting at the user-level. For example, in Raid, multicasting was simulated in the high-level communication package [BMR90]. The CPU cost for message processing is paid several times in order to send the same message to various sites.

We studied several alternatives to alleviate this problem [BMR90].The first two alternatives are based on the SE[6] protocol. The user-level SEmulticast utility is implemented on top of the SE device driver, which provides point-to-point Ethernet communication. In order to support multicast, this utility has to call the device driver for each member in the multicast group. The kernel-level SE multicast utility uses the *multiSE* device driver This device driver can send the same message to a group of destinations on the Ethernet with one system call. A third alternative uses Push[7] to implement multicasting. The Push multicasting routine is dynamically added to the kernel, where it can be interpreted by the Push machine. Finally, we experimented with physical multicasting. This method minimizes bandwidth, but demands that the multicast address be known to all members of the group, which can incur extra messages. In addition, it has to be extended in order to function in

[6]SE (Simple Ethernet) is a suite of protocols that provide low-level access to the ethernet [BMR87].
[7]Push is a facility that to add/modify kernel-level services

internetworks, or networks that do not support physical multicast. This implies the use of special algorithms for routing and maintenance of group addresses [CD85].

The simulation of multicasting inside the kernel is an important service for short-lived multicast groups. Short-lived multicast groups are frequently used in distributed transaction processing systems. Each transaction involves a different subset of sites, based on the distribution of replicas of items read or written. Multicasting to the subset of sites happens during transaction processing (to read/write or to form quorums) and during transaction commitment. In general, there are too many such subsets to define multicast groups for each possible subset. Also, the groups change so fast that the use of multicast mechanisms that require expensive group management is unacceptable.

Figure 3, compares the performance of the four multicasting methods. The programs

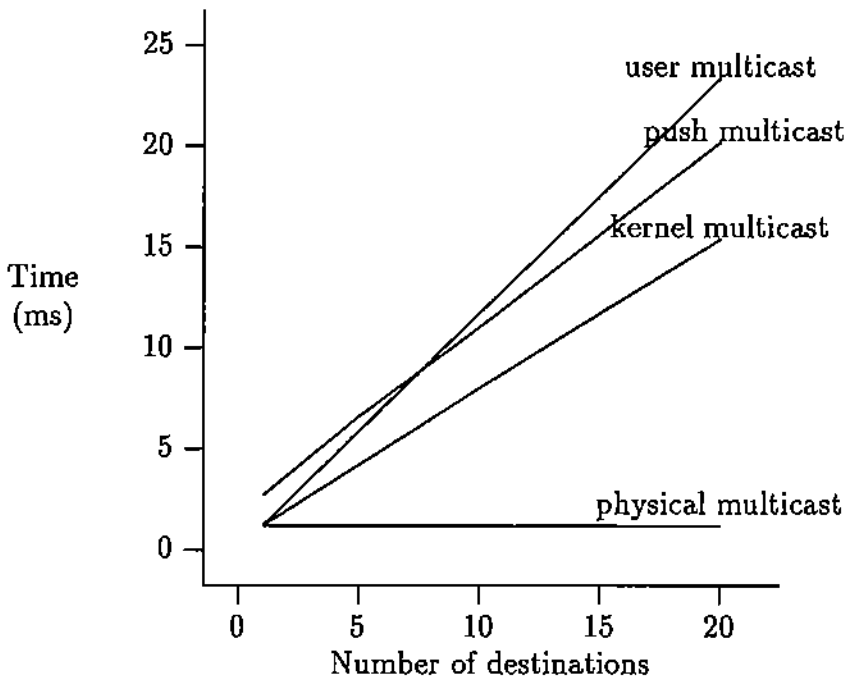

Figure 3: Multicasting cost

considered for this experiment send a 20 byte message to the set of destinations in the multicast group and return.

3.4 Experiment IV: Impact on Transaction Processing

This experiment provides the overall performance characteristics of transaction processing in the Raid system. We want to understand the impact of interprocess communication on the performance of the system. From experiments in [Maf90], we know that communication

significantly affects transaction processing. Transactions need a large number of messages and messages are expensive to process. In this subsection, we measure the times spent by Raid servers at user level, system level, and in XDR[8].

We use the DebitCredit benchmark [A+85] for our experiments. This benchmark also known as TP1 or ET1, is a simple yet realistic transaction processing benchmark. This benchmark uses a small banking database, which consists of three relations: the teller relation, the branch relation, and the account relation. The tuples in these relations are 100 bytes long and contain an integer key and a fixed-point dollar value. In addition, there is a sequential history relation, which records one tuple per transaction. Its tuples are 50 bytes long and contain a teller, branch, and account id, and the relative dollar value of the transaction. In its original form, the DebitCredit benchmark defines only one type of transaction. This transaction updates one tuple from each of the three relations and appends a logging tuple to a special sequential history relation.

For this experiment, we selected a benchmark defined by the following parameters:

- Relation size: 100 tuples.
- Hot-spot size: 20% of the tuples.
- Hot-spot access: 80% of the actions access hot-spot tuples.
- Type of experiment: close[9].
- Number of transactions: 250.
- Transaction length. Average: 6. Variance: 4
- Timeout: one second per action.
- Updates: 10% of the actions are updates.
- Restart policy: rolling restart backoff.
- Concurrency controller: two phase locking protocol.
- Atomicity controller: two phase commit protocol.
- Replication controller: ROWA.
- Concurrency level: one transaction at a time.

[8]XDR is a standard for external data representation proposed by Sun Microsystems, Inc.
[9]Close experiments maintain the concurrency level constant.

We ran this benchmark on a single-site DebitCredit database. We generated the workload with the transaction generator and repeated the experiment 30 times. The 95% confidence intervals for this sample of observations was less than 5% of the observed mean values.

Figure 4 gives a graphical representation of XDR, user, and system times for Raid servers,

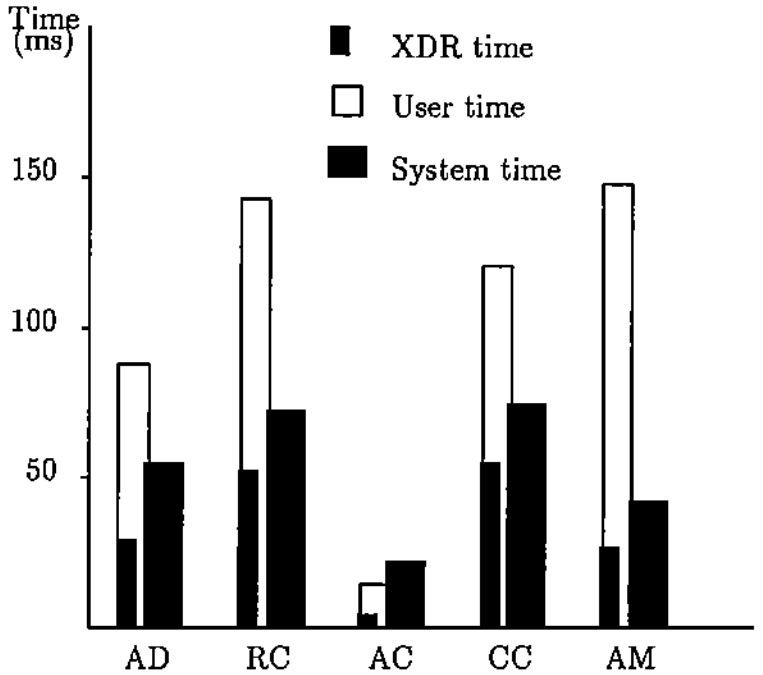

Figure 4: Raid servers' times (in milliseconds).

for an average transaction. User and system times as well as XDR times are given in seconds. User time is the time spent by the CPU processing user level code. System time is the time the CPU spends executing kernel code. XDR time is the time spent by outgoing and incoming messages in XDR routines.

3.5 Summary

In this section we summarize the problems with currently used schemes for communication software.

- General purpose interprocess communication abstractions and mechanisms are generally expensive. Although these abstractions and mechanisms are useful to support a variety of applications and users, they impose unnecessary overheads during transaction processing. For example, the socket interprocess communication abstraction is a powerful platform for the implementation of diverse communication paradigms [LMKQ89].

The mbuf memory management mechanism offers increased flexibility for the implementation of different communication protocols [LMKQ89]. However, efficiency-critical, communication-intensive applications like transaction processing, can not afford the overheads imposed by general purpose communication facilities.

- Most of the communication in distributed systems is local rather than remote. Despite this fact, local communication is implemented as a special case of remote communication in most conventional communication subsystems. In subsection 3.2 we saw that communication facilities that are specialized for the local case are simpler and more efficient.

- General purpose multicasting mechanisms require group initialization and maintenance [CD85]. Multicasting groups that are typical in distributed transaction processing are both dynamic and short lived. In this case, the overhead of group initialization can obliterate the performance advantages of multicasting. Multicast simulation is CPU and network expensive. The simulation of multicast inside the kernel reduces CPU overhead (see subsection 3.3).

- Transaction processing systems are communication intensive. There is a large traffic of messages and message processing is expensive. Most of the communication activity in distributed transaction systems is local rather than remote. As a result of this, the kernel becomes the bottleneck of the system.

In section 4, we use the results from our experiments to design and implement a distributed transaction oriented communication facility. It emphasizes light overhead, simple naming, and transaction-oriented multicast support.

4 Implementation of a Transaction-Oriented Communication Facility

In the previous section, we identified the problems with conventional communication support for distributed transaction processing in Raid. We have implemented a communication facility that addresses those problems. The design of this new communication facility makes use of the the insight gained from our experimental work in this area and of ideas proposed in [Spe86, Svo86, Che86, BALL90].

4.1 Design Principles

The design of the new communication facility for distributed transaction processing uses the following ideas learned from our experimental studies:

1. Avoid the use of top-heavy communication protocols and abstractions. The experiments in subsection 3.1 show that the socket abstraction and general purpose communication protocols unnecessarily increase communication delay. This is true for both local and remote communication.

2. Reduce kernel interaction. Transaction processing is kernel intensive [Duc89]. Our measurements show not only the large number of system calls necessary to process a transaction, but also the large amount of time servers spend at the kernel level. Kernel operations can become the bottleneck, especially in a multiprocessor environment. Efficient communication facilities will reduce system time and context switching activity (more messages processed during a time slice).

3. Minimize the number of times a message has to be copied. This is especially important for local IPC because of the intense local communication activity in a highly-structured system like Raid. Shared memory can be used for that purpose, especially for intra-machine communication.

4. Use a simple IPC memory management mechanism. Most of inter-server communication consists of short and simple control messages. Although the mbuf approach is very flexible, it has negative effects on IPC performance, as was shown in section 3.1.

5. Use the same mechanism for both remote and local communication. However, we optimize for the local case.

6. Exploit the nature of a transaction processing system in the design of its underlying communication subsystem. For a LAN, a straightforward correspondence between logical and physical communication addresses can be established. We do not need special protocols to map between logical and physical addresses anymore. Group (multicasting) addresses used during commitment time can be determined as a function of the unique transaction ID. This eliminates the need for extra messages to set up group addresses.

To minimize copying, the new communication facility uses shared memory between the kernel and user-level processes. Communication ports are uniquely identified based on the Raid addresses of the corresponding servers. The site number maps to the host address and the the triplet (Raid instance number, server type, server instance) maps to a port within the given host.

For multicasting, we use the fact that multicasting groups are formed by servers of the same type. For instance in Raid, we have two types of multicasting groups. One type of multicasting group is used for replication control. Groups of that type can be formed by RC servers only. The other type supports atomicity control and can include only AC servers. The difference between monocast and multicast addresses is in the second component of the

addresses. For monocast, we use site numbers, while for multicast, we use the transaction identifiers. A transaction id uniquely determines the members of the multicasting group for a given transaction.

4.2 Implementation Details

Our first implementation of the new communication facility is for local area network environments. It is based on shared memory (between processes and the kernel), a simple naming mechanism, and a transaction-oriented multicasting scheme. In this subsection, we discuss ports, which are the the basic communication abstraction, the naming and multicasting schemes, and the communication primitives of the new communication facility.

Ports. Processes communicate through ports. These ports reside in a memory segment shared by the process and kernel address spaces. Thus, data can be communicated between the kernel and the process without copying. This reduces the amount of copying by 50% compared to other kernel-based IPC methods (see subsection 3.2). The shared memory segment contains a transmission buffer and a set of receiving buffers. The number and length of these buffers are specified by a process at the time it opens a port. The receiving buffers form a circular queue, which is coherently managed by the kernel and the process according to the conventional producer/consumer paradigm. Associated with the transmission buffer and each of the receive buffers there is an integer, which specifies the actual length of the message. In addition, there is a counter for the number of active messages (messages that have arrived, but have not been processed by the server yet). Figure 5 shows the structure

trmlen	Transmission Buffer	Active Buffers
len1	Receive Buffer 1	
len2	Receive Buffer 2	
len3	Receive Buffer 3	
	. . . . .	
lenN	Receive Buffer N	

Figure 5: Structure of a Communication Port

of a communication port.

Naming. Within a given node, ports are uniquely identified by the triplet (Raid instance number, server type, server instance). The other component of a Raid address, site number or transaction id determines the address of the physical node for monocast or the addresses of the group of nodes for multicast respectively. In the case of Ethernet, we use only multicasting addresses for link level communication. Site numbers or transaction id's are used to build multicasting addresses by copying them into the four more significant bytes of the Ethernet address (the first bit of a multicast address has to be one).

Multicasting. During transaction processing, physical multicasting is used in the following way. While processing data requests for a given transaction, each participant site sets a multicasting address using the transaction ID as its four more significant bytes. When commitment time arrives, the coordinator uses that address to multicast messages to all participant sites. This approach takes full advantage of physical multicast, without incurring the overhead of other multicasting methods. Currently, multicast addresses are added/deleted by Raid servers. The RC adds a new multicasting address for a transaction, when it receives the first operation request for that transaction. In normal conditions, the AC deletes the multicasting address once the transaction is committed or aborted. In the presence of failures, the CC does this job as part of its cleanup procedure. In the future, we plan to manage the multicasting addresses in the communication subsystem. This will improve performance and transparency in Raid.

Communication primitives. System calls are provided to open and close a port, to send a message and to add or delete a multicasting address. There is no need for an explicit receive system call. If idle, a receiving process must wait for a signal (and the corresponding message) to arrive.

To send a message, a process writes it into the transmission buffer and passes control to the kernel. If the message is local, it is copied into a receiving buffer of the target port and the owner of the port is signaled[10]. We use the Unix SIGIO signal for this purpose. Otherwise, one of the existing network device drivers is used to send the message to its destination. The destination address is constructed as described above and the message is enqueued into the device's output queue.

When a message arrives over the network, it is demultiplexed to its corresponding port. Again, a signal alerts the receiving process about the incoming message. All this is done at interrupt time and there is no need to schedule further software interrupts.

[10]The process ID of the process that owns a port is stored in the port's data structure.

4.3 Performance Evaluation

We conducted two experiments to evaluate the performance of the new communication facility. First, we studied the performance of the basic communication mechanism at the system call level. Second, we tested the impact of the new communication subsystem on the overall performance of Raid. Here, we separately studied the contribution of the new communication subsystem to the local and remote interprocess communication activity in Raid.

4.3.1 Communication Primitives

To evaluate the performance of the basic communication primitives, we measured local and remote round trip times. Figure 6 presents the results of those measurements. For com-

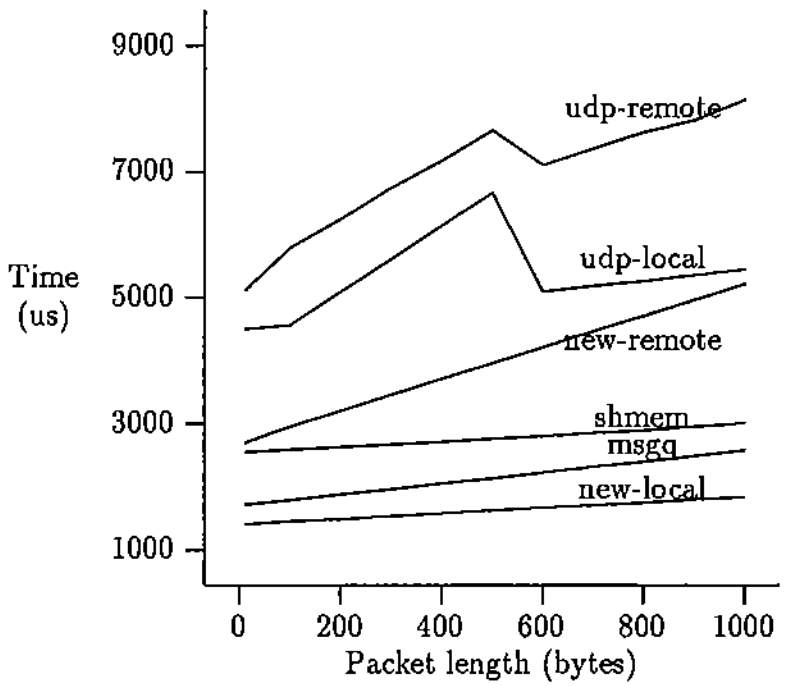

Figure 6: Round trip times (in μs)

parison purposes, we have added the corresponding times for the SunOS socket-based IPC mechanism using UDP/IP and for two local IPC methods: message passing and shared memory.

Both socket-based IPC and the new communication facility provide the same functionality in a LAN environment. Our protocol is extremely lightweight. In the local case, most of the round-trip time is context switching overhead. We measured 1.8 ms for a *round trip of context switches*, representing the context switches necessary to transmit and receive a message. (A process signals a sleeping process and goes to sleep waiting for the same signal

from the recently awakened process). We obtained better times for local round trips of messages, because of optimizations done in signaling the receiving process. In the remote case, the network device driver overhead equally affects both methods. Figure 1 shows that this overhead is significant. Despite this fact, the new communication facility achieves improvements of up to 50%. For multicasting, the performance advantages of the new communication facility become even more significant. Sending time does not depend on the number of destinations. On the other hand, multicasting time for the socket IPC method will grow linearly with the number of destinations.

Socket-based IPC does not optimize for the local case. Local round trips cost almost as much as remote ones (68–88%). In the new communication subsystem, local round trip times are only 35–50% of the corresponding remote round trips.

As in the case of shared memory and message passing, the variable component of communication delay is proportional only to the message length. For the local case, the proportionality constant is the same as that of shared memory, which is the best we can hope for in a kernel-based inter-address space interaction.

4.3.2 Impact on Performance of Transaction Processing in Raid

We carried out two experiments to test the impact of the new communication facilities on the performance of the Raid system. We wanted to see the effects of the new communication subsystem on both local and remote communication. In order to conduct these experiments, we modified the servers' structure to adapt to the new communication model. The changes are minor and are located in the main procedures [Maf90].

For these experiments, we use the experimental procedure described in subsection 3.4. In particular, we use the same benchmark. We run the benchmark on a single-site and a five-site DebitCredit databases. For the five-site database, we use the ROWA replication controller. This means that remote communication is limited to only the AC server. In addition, the benchmark contains 115 transactions that have write operations. Only those transactions need to involve remote sites in the two-phase commitment protocol.

As we discussed in subsection 3.4, most of system time is caused by communication activity. The use of the new communication mechanism allows a 62% reduction of system time for the whole Raid system. Figure 7 shows the savings in system time provided by the new communication facility for the single site case. Savings in user time are less significant [Maf90]. We also noticed a 10% reduction in context switching activity when the new communication library is used. This can be explained by the increased number of messages that can be processed during each time slice.

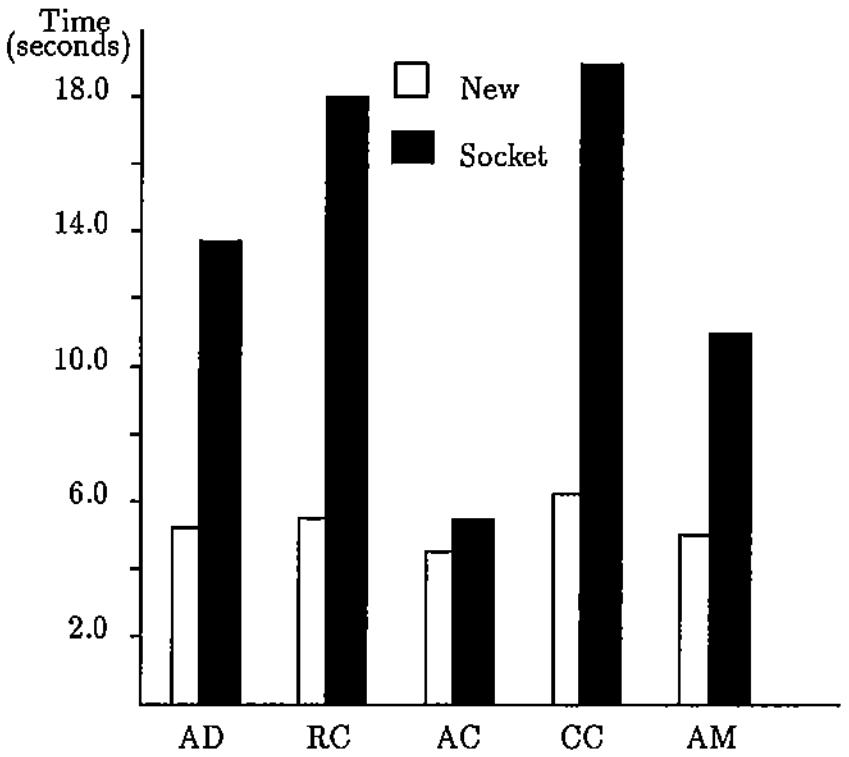

Figure 7: System Time for Raid servers (sec)

5 Conclusions and Experiences

We identified the communication services that are necessary to efficiently support a well-structured transaction processing system. Separate address spaces can be used to structure a complex transaction processing system. This can provide a natural framework to support high reliability, adaptability, and concurrency. However, that structuring approach increases cross-address space communication activity in the system. When using conventional kernel-based communication channels, the result is a communication-intensive system. Not only is the number of messages high (about 180 messages for a five-operation transaction), but also messages are expensive to process. High interaction among servers also triggers costly context switching activity in the system. Increasing availability through distribution and replication of data demands specialized remote communication. In particular, we need special purpose multicasting mechanisms. The lack of such facilities further degrades the performance of the system.

We have shown that efficient interprocess communication can make the use of address spaces a practical solution for the structuring requirements of complex distributed transaction processing system. In local area networks, the main concern is the performance of the

local cross-address space communication mechanism. For a typical transaction, about 90% of the communication activity is local. This fact has been observed for other distributed systems [Ber90]. Conventional interprocess communication facilities are optimized for the remote case [BALL90]. We addressed this problem in the design of our new communication facility. Our first prototype provides a streamlined interprocess communication service. It uses shared memory between the kernel and the server processes. This alleviates in part the demands imposed on the kernel. The use of our prototype in Raid results in a reduction of 60% – 70% of system time. Context switching activity also diminishes because more messages can be processed during the same time slice. The new communication model has a straightforward mapping between server addresses and network addresses. It also exploits the semantics of transaction processing to provide an efficient multicasting support. These two mechanisms have an effect on both system and user times. First, there is no need for explicit address translation processing. And second, the support of physical multicasting eliminates the need to simulate it at the user level.

Communication processing constitutes a significant part of user-level time. At the highest level, servers interact with each other through complex data structures. While the new communication subsystem provides efficient low level, buffer to buffer communication, it still does not take into account the high-level communication demands of a transaction processing system. Formatting data structures into buffers can become the major bottleneck of the system. We measured that XDR processing represents approximately 1/3 of user-level time. In the future, we want to explore new local communication paradigms to attack this problem. Shared memory among server processes will avoid the multiple encoding/decoding of Raid messages, reducing not only user-level time but also system time. In addition, the kernel will have to do less message processing. The lightweight remote procedure call paradigm in [Ber90] uses shared memory and a special process management mechanism to provide efficient cross address space communication. Communication under that paradigm is kernel-based. The kernel still remains as a potential bottleneck. The user level remote procedure call model in [Ber90] takes the kernel completely out a message's way. Communication and process scheduling are moved from the kernel into user-level libraries. It not only will improve the performance of the system (reduced context switching activity), but what is most important, it also will eliminate the kernel as the bottleneck of the system.

Scheduling policies in conventional operating systems do not consider high level relationships that may exist among a group of processes. Optimization of response time or throughput at the operating system level is the main driving force in those scheduling policies. That optimization may not be reflected at higher levels. In other words, conflicts may exist between the optimization criteria at the operating system and application levels. In transaction processing systems, we are interested in response time and throughput not for individual processes but for the whole system of processes. This can be achieved by introducing the concept of a system of processes as a new operating system abstraction. Scheduling can be done at two levels. At the higher level, the kernel would schedule systems of processes

as atomic entities. Internally, scheduling could be done based on internal requirements of each system. In particular, it could be based on its communication patterns. In Raid, the concurrency controller is CPU intensive and after some time it has its scheduling priority decreased. This forces CC to give up the CPU after processing only one message, even though its time slice has not expired yet.

Database applications demand the transfer of large amount of data. We believe that efficient bulk data transport services can be provided by exploiting the semantics of transaction processing systems. We need not only efficient file system support but also a closer collaboration between file, network, and communication subsystems. Remote requests for data can be handled within the kernel. This will avoid both kernel-user level interaction and multiple copying and formatting of data. Incremental remote access of data can be also supported as a result of that collaboration [PA89].

Most of our work has been based on local area networks. Wide area network transaction processing systems increase the demands on efficient remote communication. Internetwork-based systems will require more complex communication support. The multicasting scheme of our prototype cannot be used on those cases. Naming and addressing become more elaborate because of the presence of different network technologies. Finally, if the system consists of a large number of nodes, we will need to look for alternative control flows for transaction processing. One of the main objectives of those control flows has to be the reduction of the number of remote messages needed for transaction processing.

References

[A+85] Anon et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[BALL90] 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.

[Ben87] John K. Bennett. The design and implementation of distributed smalltalk. In *Proc of OOPSLA '87*, pages 318–330, Orlando, Florida, October 1987.

[Ber90] Brian N. Bershad. High performance cross-address space communication. Technical Report 90-06-02, University of Washington, June 1990.

[BFH+90] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, Srinivasan Jagannathan, and John Riedl. Design and implementation of the Raid V2 distributed database system. Technical Report CSD-TR-962, Purdue University, March 1990.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BMR87] Bharat Bhargava, Tom Mueller, and John Riedl. Experimental analysis of layered Ethernet software. In *Proc of the ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, pages 559–568, Dallas, Texas, October 1987.

[BMR90] Bharat Bhargava, Enrique Mafla, and John Riedl. Communication in the Raid distributed database system. *Computer Networks and ISDN Systems*, 1990. To appear.

[BR89a] Bharat Bhargava and John Riedl. A model for adaptable systems for transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), December 1989.

[BR89b] Bharat Bhargava and John Riedl. The Raid distributed database system. *IEEE Transactions on Software Engineering*, 15(6), June 1989.

[CD85] David R. Cheriton and Stephen E. Deering. Host groups: A multicast extension for datagram internetworks. In *Proc of the 9th Data Communication Symposium*, pages 172–179, New York, September 1985.

[Cha84] Jo-Mei Chang. Simplifying distributed database systems design by using a broadcast network. In *Proc of the ACM SIGMOD Conference*, June 1984.

[Che84] David R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating System Review*, 18(4):12–20, October 1984.

[Che86] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proc of the SIGCOMM'86 Symposium*, pages 406–415, August 1986.

[CLZ87] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A high throughput transport protocol. In *Proc of the SIGCOMM Conference*, August 1987.

[DJA88] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The CLOUDS distributed operating system: Functional description, implementation details and related work. In *Proc of the 8th Intl Conf on Distributed Computing Systems*, San Jose, CA, June 1988.

[Duc89] Dan Duchamp. Analysis of transaction management performance. In *Proc of the 12th ACM Symposium on Operating Systems Principles*, pages 177–190, Litchfield Park, AZ, December 1989.

[HPAO89] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. RPC in the *x*-kernel: Evaluating new design techniques. In *Proc of the 12th ACM Symposium on Operating Systems Principles*, pages 177–190, Litchfield Park, AZ, December 1989.

[JC89] Gary M. Johnston and Roy H. Campbell. An object-oriented implementation of distributed virtual memory. In *Proc of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 39–57, Fort Lauderdale, FL, October 1989.

[KLB89] Charles Koelbel, Fady Lamaa, and Bharat Bhargava. Efficient implementation of modularity in Raid. In *Proc of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 127–143, Fort Lauderdale, FL, October 1989.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proc of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[LMKQ89] Sammuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison Wesley, 1989.

[LZCZ86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File access performance of diskless workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[Maf90] Enrique Mafla. *Efficient Communication Support for Distributed Transaction Processing.* PhD thesis, Purdue University, December 1990.

[MSM89] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proc of the 9th International Conference on Distributed Computing Systems*, pages 129–134, Newport Beach, CA, June 1989.

[PA89] Marc F. Pucci and J. L. Alberti. Experiences with efficient interprocess communication in DUNE. In *Proc of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 349–360, Fort Lauderdale, FL, October 1989.

[PR90] David L. Presotto and Dennis M. Ritchie. Interprocess communication in the ninth edition Unix system. *Software Practice and Experience*, 20(S1):3–17, June 1990.

[Rit84] D. M. Ritchie. A stream input–output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc of the 8th Symposium on Operating Systems Principles*, pages 64–75, Pacific Grove, California, December 1981.

[SB90] Michael D. Schroeder and Michael Burrows. Performance of firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Spe86] Alfred Z. Spector. Communication support in operating systems for distributed transactions. In *Networking in Open Systems*, pages 313–324. Springer Verlag, August 1986.

[STP+87] Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger, Dan Duchamp, Richard Draves, Dean S. Daniels, and Joshua J. Block. CAMELOT: A distributed transaction facility for MACH and the Internet — An interim report. Technical Report CMU-CS-87-129, Department of Computer Sciences, Carnegie Mellon University, June 1987.

[Svo86] Liba Svobodova. Communication support for distributed processing: Design and implementation issues. In *Networking in Open Systems*, pages 176–192. Springer Verlag, August 1986.

[VM90] Paulo Veríssimo and José Alves Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proc of the 9th Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, Alabama, October 1990.

[vRvST88] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the world's fastest distributed operating system. *Operating System Reviews*, 22(4):25–34, October 1988.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc of the 11th Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987.

[Zwa85] Willy Zwaenepoel. Protocols for large data transfers over local networks. In *Proc of the 9th Data Communications Symposium*, pages 22–32, Whistler Mountain, British Columbia, Canada, September 1985.