Evolution of a Communication System for Distributed Transaction Processing in Raid

Bharat Bhargava, Yongguang Zhang
Purdue University
Enrique Mafla
Universidad San Francisco, Quito

ABSTRACT: This paper identifies the basic services required from a communication subsystem to support transaction processing in a distributed, reliable, reconfigurable, and replicated database environment. These services include multicasting, remote procedure calls (RPC), inexpensive datagram services, and efficient local interprocess communication (IPC). We show the evolution of the various versions of Raid communication software and our experience with them. We show how several ideas such as lightweight protocols, simple naming, memory mapping and shared memory, physical multicasting, direct control passing, and adaptability fulfill the needs for transaction processing. We present performance data on these ideas and study their impact on transaction processing. We also use the results of these studies to design and implement a new communication scheme. This scheme has reduced the overhead of communication in Raid distributed database system 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

Many applications consider the transaction as a unit of work. Transaction processing systems must manage such functions as concurrency, recovery, and replication. One way to increase the software modularity is to have the separate components of such a complex system execute in separate address spaces [Ber90, MB91a]. Enforced by hardware, this structuring paradigm increases the reliability, failure isolation, extensibility, interoperability, and parallelism of the software. However, applications that require transaction processing in distributed database systems demand high performance. Performance requirements cannot be satisfied without efficient interprocess communication (IPC) services provided by the operating system. Systems with poor IPC are forced to sacrifice either structure or performance [vRvST88, BALL90].

The *micro-kernel* or *backplane* paradigm for structuring distributed systems offers several advantages. Efficiency in local interprocess communication can be achieved by moving some critical kernel services to the user level. On the other hand, specialized kernel-level communication facilities can improve the performance of remote communication by eliminating many unnecessary layers. Local area network (LAN) differs from wide area network (WAN) in many aspects. Some are fast and reliable, such as Ethernet and Fiber Distributed Data Interface [Ros89], others are often slow and unreliable, such as the Internet. One way to take full advantage of the local networks is to use different mechanisms for different environment. By integrating different mechanisms in adaptable approach, we can build a communication system that is able to self-adjust to many different environments.

1.1 Objectives of our Research

The objective of this paper is to present a synopsis of the many ideas used in the design and development of an efficient communication support for distributed transaction processing systems in conventional architectures[1]. We present the problems encountered, the overhead of schemes available to us, the benefits drawn from the related research efforts, the performance improvement in the new implementations, the experiences from our research, and finally the effects on the transaction processing in the context of an experimental distributed database system [BR89].

We observed that the address space paradigm can provide a natural platform for the support of extensibility, modularity, and interoperability. It can also increase opportunities for parallelism, which improves the concurrency, reliability, and adaptability. The complete system (including the kernel, operating system services, and applications) can be decomposed into a set of smaller and simpler modules. These modules are self-contained and interact with each other via well-defined interfaces. First, the concurrency is enhanced because the processes are the schedulable units of the system. The input/output will not block the whole system, but only the module that handles the interaction (e.g. the disk manager, or the communication manager). The other modules of the system can still run concurrently. Second, reliability improves 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 in a dynamically reconfigurable system.

Our observation are summarized as follows: Some generic abstractions (e.g. the Unix socket abstraction) which feature transparency are expensive. Many improvements can be made to decrease communication costs: memory mapping can reduce kernel interactions and copying, physical multicasting can be efficiently implemented, a lightweight protocol for LAN can improve the communication performance, direct control passing and shared memory can be useful in transmitting com-

1. By conventional architectures, we mean virtual-memory, single-processor machines with no special hardware support for interprocess communication. (Some main frame computers have hardware assistance for IPC where more than one address space can be accessed at the same time.)

plex data structures. Our implementation shows that different communication models can be integrated under a unified interface to support many types of communication media.

1.2 The Raid Distributed Database System

Raid is a distributed transaction processing system specifically designed for conducting experiments [BR89] on transaction management schemes. Distributed transaction processing involves following functions:

- Transparent access to distributed and replicated data.
- Concurrency control.
- Atomic commitment control.
- Consistency control of replicated copies.
- Transparency to site/network failures (reconfiguration capabilities).

In Raid, each of these functions is implemented as a server. Each server is a process in a separate address space. Servers interact with each other through a high-level communication subsystem. They can be arbitrarily distributed over the underlying network to form logical database sites. Currently there are four subsystems for each site: an atomicity controller (AC), a concurrency controller (CC), an access manager (AM), and a replication controller (RC). For each user in the system there is an action driver server (AD), which parses and executes transactions. The AC coordinates the global commitment/abortion of a transaction. The CC enforces the local serializability. The AM controls the access to the physical database. The RC ensures the consistency of replicated copies. The processing of a transaction in the system is described in detail in [BR89, BFH+90].

The experimentation with transaction processing algorithms requires a clear assignment of the functionality to each server, a flexible and well-defined way to describe the interfaces among different servers, and efficient communication support. Raid is designed to meet these requirements. Logically, each Raid server is uniquely identified with the 4-tuple: Raid instance number, Raid virtual site number, server type, server instance. Abstractions such as multicasting and RPC can be used. The communication between the servers is intensive. For example, a five operation (read or write) transaction in Raid

requires over thirty local messages in the coordinating site, eight local messages in other participating sites, and four remote messages per site.

1.3 Experimental Studies and the Benchmark

The experiments described in this paper were conducted in the Raid laboratory, which is equipped with five Sun 3/50s and four Sun 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. Most of the experiments were done on Sun 3/50s, one of which is configured with a special microsecond resolution clock[2] that is used to measure elapsed times.

The details of the experimental infrastructure of Raid can be found in [BFHR90a, BFHR90b]. Here we briefly describe the benchmark that we used throughout our experimental studies.

We used the DebitCredit benchmark [A+85] for our experiments. Known as TP1 or ET1, it is a simple yet realistic transaction processing benchmark that is well-accepted. 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 the attributes with id's of a teller, a branch, and an account, and the relative dollar value specified in 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.

To obtain a greater variety of transaction streams, we extended the benchmark. In our experiments, we used the following parameters:

- Relation size: 100 tuples.
- Hot-spot size: 20% of the tuples.

2. This timer has a resolution of up to four ticks per microsecond. The overhead to read a timestamp is approximately 20 s.

- Hot-spot access: 80% of the actions access hot-spot tuples. (That is, 80 of the access focuses on 20% of the data items.)
- Type of experiment: closed. (In a closed experiment the multiprogramming level is fixed. When one transaction completes another one is started. The opposite is an open experiment, in which the transaction inter-arrival gap can be varied to control the system load.)
- Number of transactions: 250.
- Transaction length: the average number of the read/write operations per transaction is 6. The variance is 4.
- Timeout: one second per action.
- Updates: 10% of the operations are updates. (However, it is much higher in terms of transaction, since one transaction may consist of several read and write operations.)
- Restart policy: rolling restart backoff. (A transaction that is aborted in the processing must be restarted sometime later. In rolling restart backoff, the delay of restart is set to the average response time of the last few transactions.)

We used the following algorithms to implement the Raid transaction processing:

- Concurrency controller: two phase locking protocol.
- Atomicity controller: two phase commit protocol.
- Replication controller: ROWA (Read One Write All) protocol.
- Concurrency level: one transaction at a time.

This paper is organized as follows. The next section (2) introduces the details of the version 1 of the communication software. Section (3) discusses the evolution of the version 2 of the software and identifies the problems and suggests several guidelines for possible improvements. The design and the performance of the version 2 software and its impact on Raid transaction processing are also presented. Section (4) lists our experiences with the version 2 and suggests several steps towards the evolution of the version 3 of the communication software. Section (5) presents an adaptable approach to the software and demonstrates how different mechanisms can be used for different environments. The three versions of the software are compared. Section (6) discusses other work in related areas. Finally section (7) presents our conclusions and experiences.

2. The Raid Communication Subsystem Version 1

The development of Raid emphasizes on the structural construction of Raid servers. The main purpose of the communication subsystem version 1 was to provide a clean, location independent interface between the servers [BMR91].

2.1 Structure

The *Raidcomm V.1* was implemented on the top of the SunOS socket-based IPC mechanism using UDP/IP (User Datagram Protocol/Internet Protocol). To permit the server interfaces to be defined in terms of arbitrary data structures, XDR (Sun's eXternal Data Representation standard) is used. A high-level naming service called the oracle was added. The oracle is a global server and is not attached to any particular Raid site. The oracle responds to a request for the address of a server by returning the UDP/IP address. The oracle also provides a notification service that sends a message to all interested servers when another server enters or leaves the system. All these messages are handled automatically by the communication software at each server, which also automatically caches the addresses of other servers as they are received from the oracle. Facilities for high-level multicasting, messages of arbitrary length, etc. were added later.

Figure 1 shows the layering of the *Raidcomm V.1* package.

Raid servers communicate with each other using high-level operations such as `AD_StartCommit_AC()`, which is used by the AD to start commitment processing. At this level, servers pass to each other the machine dependent data structures. XDR marshals those data

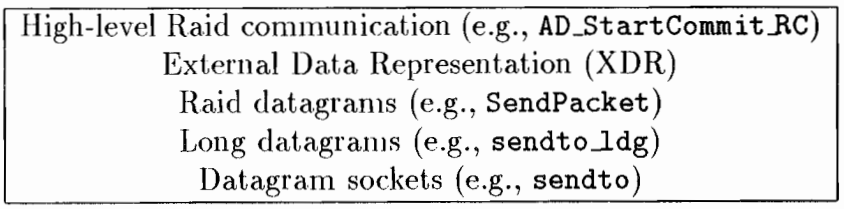


Figure 1. Layers of the Raid communication package version 1.

structures into a buffer, using a machine independent data representation. Raid datagrams are similar to UDP datagrams, but specify their destination with a Raid address. LDG (Long Datagram) has the same syntax and semantics as UDP, but has no restriction on packet length. The fragment size is an important parameter to LDG. Normally we use fragments of 8 Kbytes size, which is the largest possible on our Suns. Since IP gateways usually fragment messages into 512 byte packets, we have developed a version of LDG with 512 byte fragments. This allows us to compare the kernel-level fragmentation in IP (Internet Protocol) (if LDG fragments are larger than 512 bytes) with the user-level fragmentation in LDG (if LDG fragments are smaller than 512 bytes).

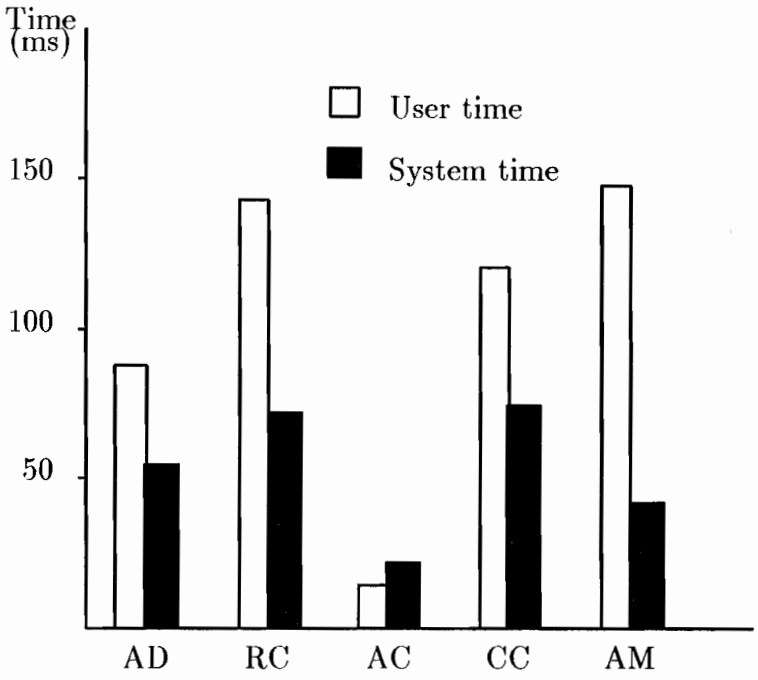
Raidcomm V.1 used a conventional approach to employ the existing communication facilities. As has been presented in [MB91a] and discussed in the next section, the conventional scheme provides unacceptable overhead.

2.2 Performance of Raidcomm V.1

We evaluated the performance characteristics of the transaction processing in the Raid system by running transactions based on the benchmark described in section 1.3. The objective was to study the impact of *Raidcomm V.1* on the performance of the system. We measured the times spent by Raid servers at the user level and the system level.

We ran this benchmark on both a five-site DebitCredit database and 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.

The system ran well in the five-site database case. However, there are performance problems in the single-site case. Figure 2 gives a graphical representation of the average user times and the average system times to process a transaction in each Raid servers in the single-site case. The user and system times are given in milliseconds. The user time is the time spent by the CPU while processing the user level code. The system time is the time that the CPU spends while executing kernel code. We recorded the average number of messages sent and received for each transaction to be 168.



Legend: AC = atomicity controller
CC = concurrency controller
AM = access manager
RC = replication controller
AD = action driver

Figure 2. Raid servers' times (in milliseconds).

Communication related processing takes place at both the user and system levels. Most of the system time is spent in the interprocess communication. Raid servers make two types of system calls: synchronous disk I/O (for logging) and interprocess communication. For each transaction, we can assume that synchronous disk I/O consumes less than 22 milliseconds, since AC is the only server that invokes this kind of system call and its system time is 22 milliseconds (this includes the time necessary to process AC's 6 interprocess communication system calls). We have a total of 168 interprocess communication system calls (send and receive) and the total system times for each

transaction is 267.4 milliseconds. This implies that more than 92% of system times are spent on communication, and each message-sending and message-receiving system call takes an average of 1.5 milliseconds.

3. *Evolution of the Raid Communication Subsystem Version 2*

The performance analysis in the previous section confirms that the distributed transaction processing system is communication intensive and the message processing is expensive. We have reported a series of experiments and observations [BMR91, MB91a] on the facilities available to us for the design of Version 1 of the software. These experiments helped us in understanding the weaknesses of these facilities and guided us towards the development of the new communication services. We briefly identify the issues of IPC, multicasting, and naming.

3.1 Problems with the Interprocess Communication

We have 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, the communication protocols (UDP, IP, and Ethernet), and the interrupt processing. We measured the round-trip performance of several local interprocess communication facilities available on SunOS. We investigated several mechanisms including the double message queue, the single message queue[3], the named pipe, the shared memory with semaphore, and the UDP sockets in both the Internet and UNIX domains [MB91b, MB91a].

We observed that the conventional communication schemes, such as UDP that are used in *Raidcomm V.1,* are not suitable for complex

3. The message queue is a S system V Unix interprocess communication scheme. Two processes in the same machine can communication by sending and receiving messages through a shared message queue. The double message queue model uses two message queues between every two local processes, one for each direction. The single message queue model uses only one message queue for both directions.

distributed transaction processing systems. Even though many abstractions and mechanisms are useful to support a variety of applications and users, messages have to pass through several unnecessary layers of the communication subsystem. For example, the socket abstraction and `mbuf` memory management mechanisms are irrelevant layers for the transaction processing [MB91b, MB91a]. To overcome these problems, a simple IPC memory management mechanism should be used. Both virtual/layered protocols in *x*-kernel [PHOR90] and VMTP [Che86] provide support to avoid such an overhead (see section 6).

We observed that the benefits of using special-purpose methods for the local interprocess communication are significant. We measured the round trip time for a null message for message queues at 1.7 milliseconds. But for the Internet domain UDP socket, it took 4.4 milliseconds. It still took 3.6 milliseconds by the UNIX domain UDP socket [MB91a]. By using the memory mapping to reduce the kernel interaction and copying, we can provide more efficient transaction-oriented local interprocess communication.

We also found that the name resolution can become an expensive and complicated process. In general, we can have three different name spaces: the application name space, the interprocess communication name space, and the network name space. We use a special protocol to map Raid names into interprocess communication addresses (UDP/IP addresses). These addresses have to be mapped into the network addresses (e.g. Ethernet addresses) via a second address resolution protocol [MB91b]. However, one way to break down the layers in a LAN environment is to establish a straightforward correspondence between logical and physical communication addresses.

3.2 Strategies for Implementing Multicast Communication

The replication and commitment protocols require a mechanism for multicast messages. Many operating systems do not provide any multicasting facility. Applications simulate the multicasting at the user-level. In *Raidcomm V.1*, the multicasting was simulated in the high-level communication package [BMR91]. The CPU cost for the message processing is paid several times in order to send the same message to various sites.

We studied several alternatives to alleviate this problem [BMR91]. The first one was called physical multicasting. Physical multicasting is attractive since it minimizes the network bandwidth requirement. However, it 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 to function in the interconnected networks, or networks that do not support physical multicasting. This implies the use of special algorithms for the routing and maintenance of group addresses [CD85]. A second mechanism that is more flexible is to build multicasting at the user level. It is implemented so as to call the device driver for each member in the multicast group. The third method is to provide multicasting support inside the kernel. One system call allows the sending of the same message to a group of destinations (on the Ethernet). Therefore it is much more efficient than the user-level multicasting. Finally, the fourth alternative is to use the Push[4] facility to add multicasting routine into the kernel dynamically.

We built a Simple Ethernet (SE)[5] device driver to provide a point-to-point Ethernet communication. The user-level multicasting utility is implemented on top of the SE device driver. We implemented the kernel-level multicasting as the *multiSE* device driver. We measured the performance of the four multicasting methods by sending a 20-byte message to a set of destinations in the multicast group. The results are shown in Figure 3.

The user-level multicasting is slower than the kernel multicasting because it invokes a system call for each destination. Although Push multicasting eliminates the system call overhead, the interpretive execution of Push code by the Push machine makes it even slower than the user-level multicasting when the number of the destinations is small.

The simulation of multicasting inside the kernel is an important service for short-lived multicast groups. Short-lived multicast groups are frequently used in the distributed transaction processing systems. Each transaction involves a different subset of sites, based on the dis-

4. Push is an experimental facility developed in the Raid laboratory [BMR89]. It provides a software mechanism to add kernel-level services dynamically, and an interpretor inside the kernel to execute user-level code.
5. SE (Simple Ethernet) is a suite of protocols that provide low-level access to the ethernet [BMR91].

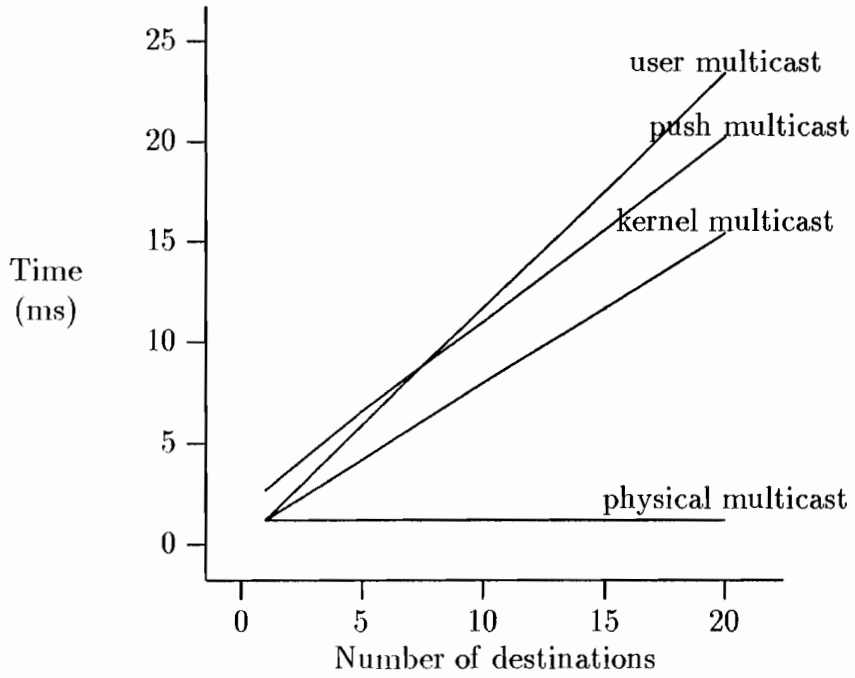


Figure 3. Multicasting cost

tribution of replicas of items read or written. Multicasting to the subset of sites is needed (to read/write or to form quorums) and during the transaction commitment. In general, there are too many such subsets to define one multicast group for each of them. Also, the groups change so quickly that the use of any multicast mechanism that requires expensive group management is unacceptable. Thus the physical multicasting for such short-lived multicast groups is not easily possible.

3.3 Guidelines for Improvements

The following ideas and guidelines were used to implement the second version of the communication software.

1. Avoid the use of heavy-overhead communication protocols and abstractions. The socket abstraction and general purpose communication protocols unnecessarily increase communication delay. This is true for both local and remote communication.

2. Reduce the 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 that servers spend at the kernel level. Kernel operations can become the bottleneck, especially in a multiprocessor environment. Efficient communication facilities will reduce the system time and the 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 the inter-server communication consists of short and simple control messages. Although the `mbuf` approach in Unix is flexible, it has negative effects on the IPC performance.
5. Exploit the characteristics of the 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 these addresses. Group (multicasting) addresses used during the commitment time can be determined as a function of the unique transaction ID. This eliminates the need for extra messages to set up the group addresses.

Our implementation of *Raidcomm V.2* is appropriate for the local area network environments. It is based on the memory mapping[6], a simple naming mechanism, and a transaction-oriented multicasting scheme.

3.4 Implementation of RaidComm Version 2

The development of *Raidcomm V.2* is done on the platform of SunOS. We modified the SunOS kernel and added several new modules. Here we briefly describe the most important ideas in our implementation.

6. A segment of kernel address space is mapped into the address space of a user process. This allows the user process and the kernel to share the same segment of the physical memory.

Communication ports are uniquely identified based on the Raid addresses of the corresponding servers. The site number maps to the host address and the triplet Raid instance number, server type, server instance maps to a port within the given host. Both mappings are one to one.

For multicasting, we use the fact that multicasting groups are formed by the servers of the same type. For instance in Raid, we have two types of multicast groups. One type of the multicast group is used for the replication control and is formed by RC servers only. The other type supports atomicity control and includes 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. Figure 4 shows the architecture of the *Raidcomm V.2* package.

Ports reside in a memory segment accessible by both the process and the kernel address spaces. Thus, data can be communicated between them without copying. This reduces the amount of copying by 50% compared to other kernel-based IPC methods. Figure 5 shows the structure of a communication port.

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 determines the address of the physical node if

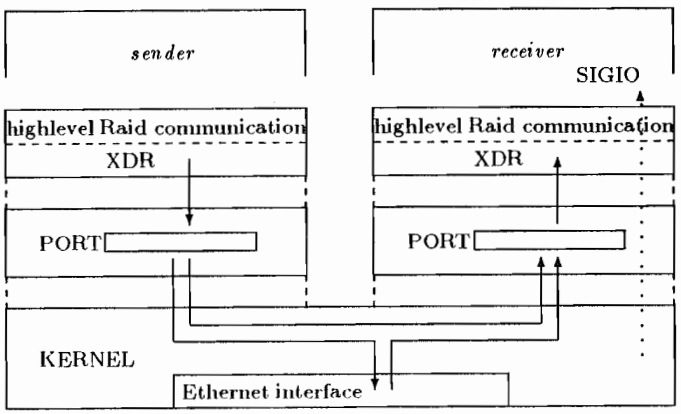


Figure 4. The structure of *Raidcomm V.2*

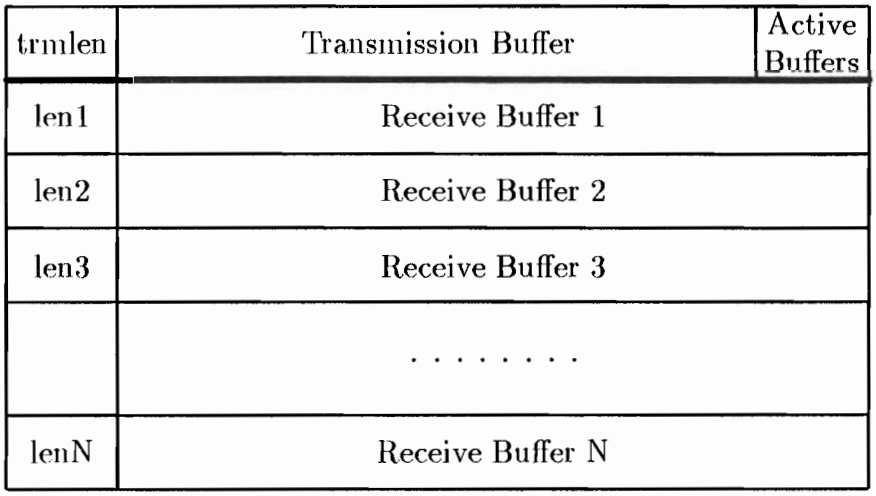


Figure 5. The structure of a communication port

it is a site number, or the multicast addresses of the group of nodes if it is a transaction ID. For the Ethernet, we use only multicast addresses for link-level communication. Site numbers or transaction id's are used to build multicast addresses by copying them into the four more-significant bytes of the Ethernet address.

During transaction processing, physical multicasting is used. While processing the requests for a given transaction, each participating site sets a multicast address using the transaction ID as its four more-significant bytes. When commitment is to be done, 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.

System calls are provided to open and close a port, to send a message and to add or delete a multicast 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, the kernel copies it to a receiving buffer of the target port and the owner of the port is signaled[7]. We use the Unix SIGIO signal for this purpose. Oth-

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

erwise, 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. If the receiving port is full, or network is congested, the send operation will abort and return error to the sender.

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.

3.5 Performance of the Communication Primitives

The basic communication primitives used in *Raidcomm V.2* have a better performance than those in *Raidcomm V.1* for both the local and the remote round trip times. Figure 6 presents our measurement for both the local and remote round trip times. For comparison, we have added

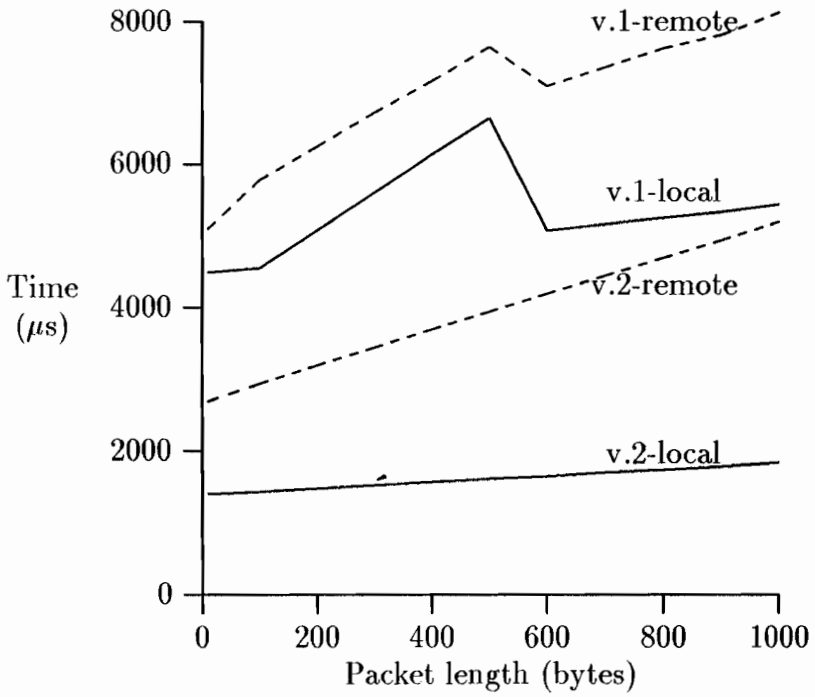


Figure 6. Round trip times (in

the corresponding times for the communication primitives used in *Raidcomm V.1* —the SunOS socket-based IPC mechanism using UDP/IP.

Both socket-based IPC and the new communication primitives used in *Raidcomm V.2* provide the same functionality in a LAN environment, and both are equally affected by the significant network device driver overhead. Despite this fact, this new communication facility achieves improvements of up to 50%. For multicasting, the performance advantages of these new primitives 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 costs are close to remote ones (68-88%). In *Raidcomm V.2*, local round trip times are only 35-50% of the corresponding remote round trips.

3.6 Impact of RaidComm V.2 on Raid

We carried out two experiments to test the impact of *Raidcomm V.2* on the performance of the Raid system. We wanted to see the effects on both local and remote communication. For these experiments, we used the benchmark described in section . We ran the transactions on a single-site and a five-site DebitCredit database. For the five-site database, we used the ROWA (Read Once Write All) replication method, which means that remote communication was limited to only the AC server. In addition, the benchmark contained 115 transactions that had write operations. Only those transactions needed to involve remote sites in the commit protocol.

As we discussed in section 2.2, most of the system time is caused by communication activity (above 92%). By using memory mapping to reduce the copying, using simple naming schemes to reduce the name resolution overhead, and using transaction-oriented multicasting scheme to reduce extra messages, the system time used in *Raidcomm V.2* was significantly reduced. Figure 7 shows the saving in the system times for each transaction in the single site case. (c.f. Figure 2.)

However, when we examine the user times spent in both *Raidcomm V.1* and *Raidcomm V.2*, we find that the savings in the user times are less significant (Figure 8 vs. Figure 2). This is because our

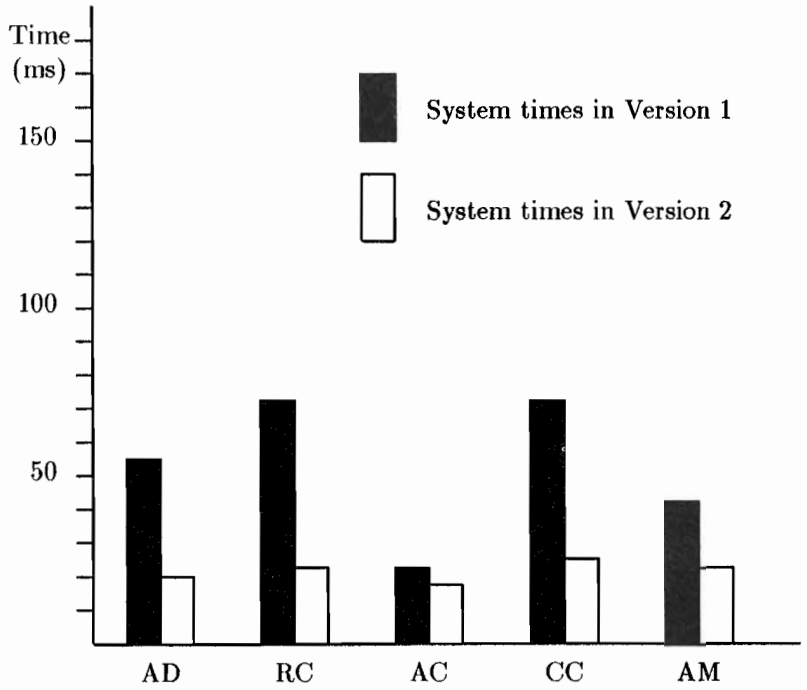


Figure 7. System time for Raid servers (ms)

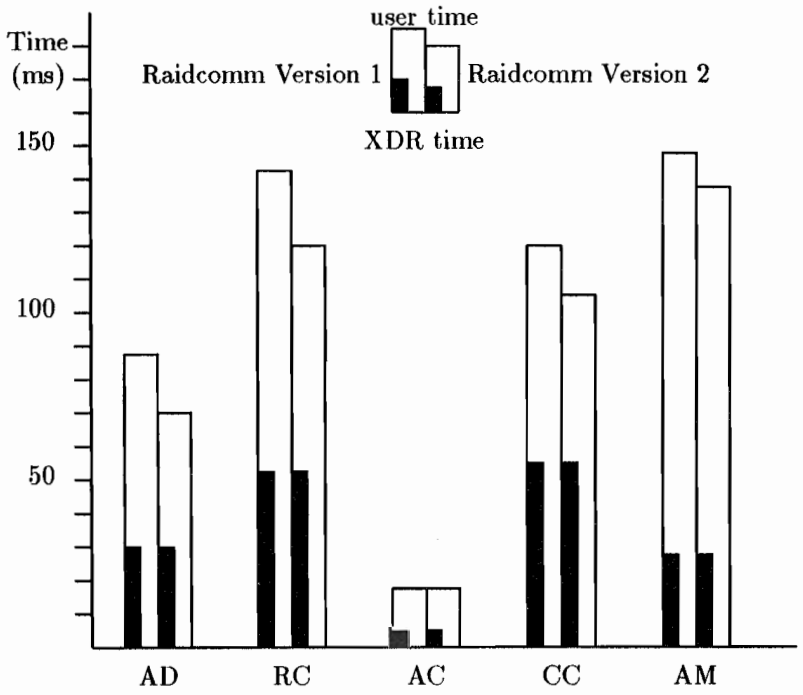


Figure 8. Average user times for a transaction in a single machine (ms)

ideas were initially focused only on the communication delays that contribute to the system time.

A closer look at the user-level of the communication subsystem reveals that XDR is responsible for approximately 1/3 of the user-level time. In the *Raidcomm V.2*, while the system time is reduced by an average of 62% for the whole Raid system, the user-level time does not drop significantly [MB91b]. XDR has become a new bottleneck in the system.

4. *Problems, Improvements and Evolution of Raid Communication Subsystem Version 3*

Our experiences with *Raidcomm V.2* were gained while conducting the experiments in Raid. This communication facility solves several problems in several conventional schemes that were mentioned in section 3.1. It provides better performance and fits the requirement of distributed transaction processing. However, it provides a new inspection point for us to understand and further examine the requirements of a distributed transaction processing system for advanced applications. Some problems that were hidden behind the ones that have been resolved are now exposed. In this section, we focus on these problems.

4.1 Problems with XDR

XDR is a widely used external data representation standard, which is used to format data between the application data structures and the transportation data structures. In future applications, several types of complex data objects will be manipulated [DVB89]. In such systems, the servers will interact with each other through complex data structures in the highest level. This requires the underlying communication subsystem to provide a cheap transportation mechanism for these structures. The *Raidcomm V.1* and *Raidcomm V.2* did not take into account such high-level communication demands. The transportation data structures are usually bounded linear buffers. Data must be coded into the sending buffers before transfer, and must be decoded from the receiv-

ing buffers to application data space. Such formatting can become a major bottleneck in the system.

Although the physical media of inter-machine communication enforces such formatting, it is never a necessity in local communication between two processes in the same machine. For example, we can build a local communication channel without such multiple encoding/decoding, if we make full use of the shared memory. Unlike lower-level buffer-based communication resources, which are one-dimensional, and can usually be only accessed as a whole, the shared memory segments are multi-dimensional randomly addressed storage resources just like the main memory. Moreover, because the transfer of large complex data object is often limited to the servers in the same site, we can expect better performance if we eliminate XDR in the local communication.

4.2 Problems with Context Switch and Scheduling

Scheduling policies in the conventional operating systems do not consider high level relationships that may exist among a group of processes. Optimization for response time or throughput at the operating system level is the driving force in such scheduling policies. This optimization may not be reflected at higher levels. In other words, conflicts may exist between the optimization criteria at the operating system and at the application levels. For example, we observed the following scenario in Raid: Since the concurrency controller (CC) is CPU-intensive, Unix will decrease its scheduling priority after some time. This forces CC to give up the CPU after processing only one message, even though its time slice has not expired yet. This reduces the performance of the whole system. In transaction processing systems, we are interested in the response time and throughput not for individual processes but for the whole system of processes. The underlying operating system should provide some way for the application to negotiate the CPU allocation.

Few operating systems can let the high-level application share this monopolized power. Context switches are often caused by preemptive events. The sender and the receiver process have to collaborate under some high-level mechanism provided by the operating system to have

the control transferred. In Unix, for example, the receiver process often goes to sleep to lower its priority. It is then put in a list and wait for the event that will cause the CPU to be granted to it. Samples of such events are I/O, signals, etc. However, these kinds of control passing facilities are expensive.

The use of shared resources (such as shared memory segment) raises the issue of synchronization. The synchronization is enforced by operating systems in the form of entities such as critical resources, semaphores, etc. However, the context switching caused by such types of synchronization is expensive.

We conducted a series of experiments on context switches. We used the idea of direct control passing in the experiments. It is a system call that allows to specify the CPU explicitly the name of the process that should run next. This special kind of scheduling is very similar to the hand-off scheduling in Mach [Bla90], except that we have extended it to schedule the ordinary processes in Unix. Figure 9 shows the results of our experiments. The measurements are for the round trip context switch times between two processes. The round trip of context switch is formed in the following way: process A makes a system call to relinquish the CPU and to cause process B to run, then process B does nothing but invokes a corresponding system call that gives up the CPU immediately and causes the control to be switched back to process A. We measured the real time of such a round trip in the enhanced SunOS caused by the original semaphore P-V operation (a system V Unix IPC facility), the original signal (SIGIO) passing, and the newly added direct control passing mechanism. We note from

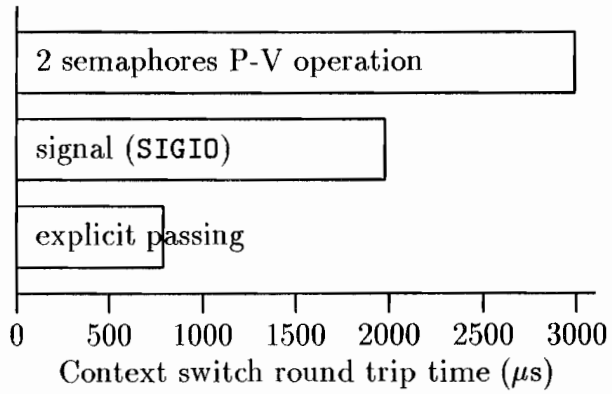


Figure 9. The performance of context switch

the figure that the context switch overhead can be reduced if we adopt a light-weight scheduling scheme, such as the direct control passing mechanism.

4.3 Design of RaidComm Version 3

The basic design behind *Raidcomm V.3* is a direct result of the problems we experienced with *Raidcomm V.2*, and the observations we made during the evaluation of its performance. The implementation of this version of the Raid communication subsystem is for the communication local to one machine. It is based on a combination of the direct control passing mechanism and the shared memory. The implementation is also based on SunOS. New communication modules will be added to the kernel.

4.3.1 Structure

Communication takes place in a shared-memory segment between two servers. Many systems use the shared memory to reduce message copying, and hence result in a higher data-coupling between two processes. However, we use shared memory to eliminate the unnecessary data conversion.

Figure 10 shows the structure of the design of *Raidcomm V.3*.

To avoid the unnecessary data coupling, every pair of processes that communicate has a dedicated segment. When two processes set up the communication, a memory segment is allocated by the kernel and attached to the address spaces of both processes.

4.3.2 Communication Primitives

System calls are provided to open and close a communication channel between two processes, to send and to receive a message, and to pass control to a specified process. The *send* system call writes the message into the shared communication segment, and passes the control directly to the receiving process. The *receive* system call puts the calling process into a special sleep-for-message state if there is no outstanding message. The *control-passing* system call bypasses the Unix scheduling by putting the specified process into the beginning of the run queue, even if it has much lower priority. This system call provides a way for

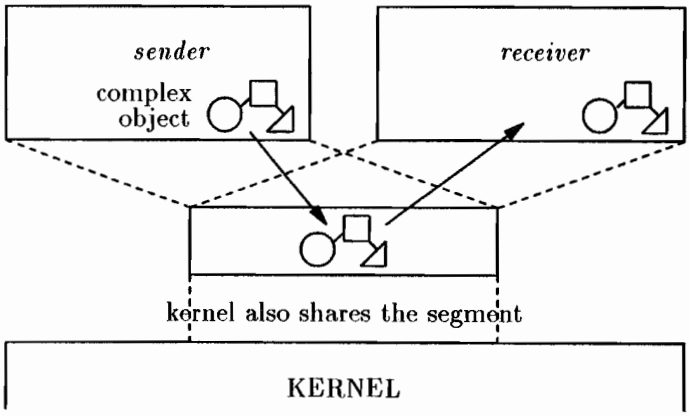


Figure 10. Local communication based on shared memory

the application processes to balance its needs with the needs of the rest of the system. In *Raidcomm V.3*, most of the kernel services involved in sending and receiving a message are either moved to the user level or bypassed.

4.3.3 Multicast in a Local Host

When a process sends a message to several processes at the same time, the message will be copied to the corresponding shared memory segments, and the control is passed one by one to all the destinations explicitly, before the sender regains control of the CPU.

4.3.4 Performance

We conducted a series of experiments to compare the processing costs for the same set of high-level Raid messages in three different versions of the Raid Communication software. We measured the CPU times spent in sending various messages in each version. The messages are restricted to local communication only. The results of the measurements are shown in Figure 11.

We can see that the improvement achieved by eliminating data copying and XDR is remarkable, and the performance is roughly uniform for all Raid messages. This is the best that can be expected for a kernel-based cross-address-space communication facility.

Figure 12 shows that the combination of the shared-memory and the explicit context switch provides shorter message round trip times.

Message († multicast dest = 5)	Length (bytes)	*Raidcomm*		
		V.1 (μs)	*V.2* (μs)	*V.3* (μs)
SendNull	44	2462	1113	683
MultiNull †	44	12180	1120	782
SendTimestamp	48	2510	1157	668
SendRelationDescriptor	76	2652	1407	752
MultiRelationDescriptor †	72	12330	1410	849
SendRelation	156	3864	2665	919
SendWriteRelations	160	3930	2718	1102

Figure 11. Performance comparison of the communcation libraries

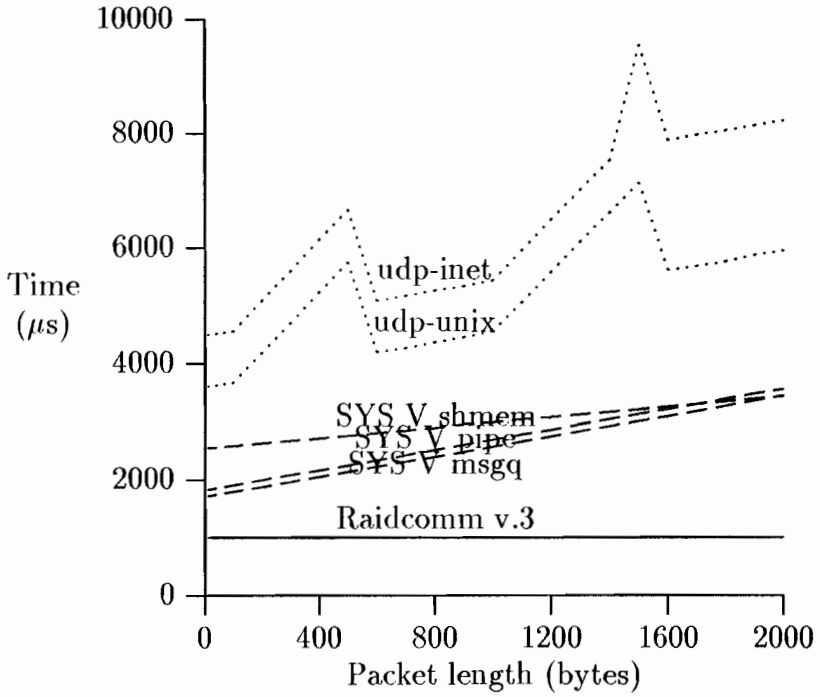


Figure 12. Comparison of message round trip times

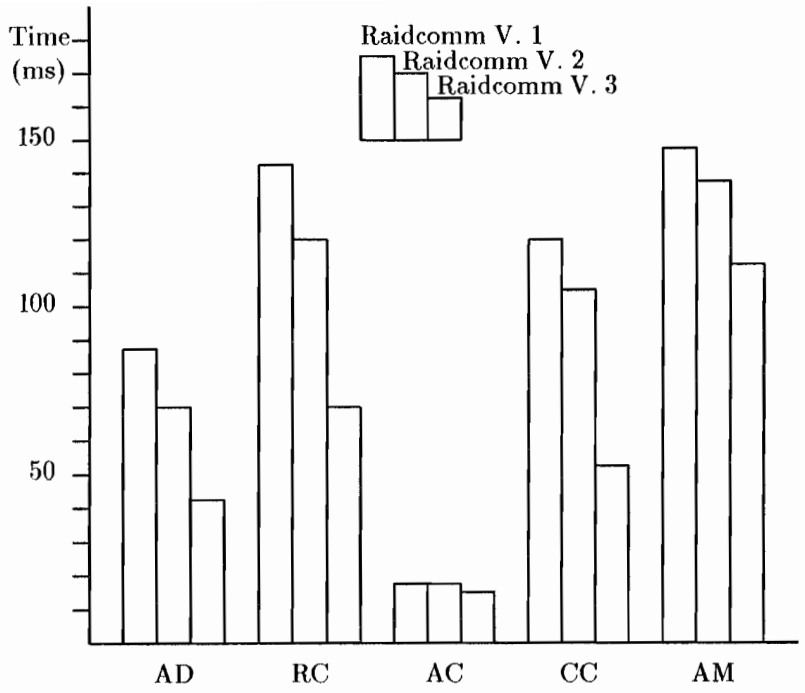


Figure 13. User times for Raid servers (ms)

For comparison, measurement of UDP sockets and SunOS interprocess communication are included. The peaks of UDP sockets reflect the special memory allocation policy used by SunOS [MB91b].

4.3.5 Impact of Raidcomm V.3 on Raid

While the system time was reduced in *Raidcomm V.2*, the user time is reduced significantly in *Raidcomm V.3*. We ran the same benchmark (see section 1.3) in one-site DebitCredit databases. Figure shows the saving in the user time caused by the further reduction in data copying and the elimination of XDR.

5. *Adaptable Approach to Communication Software*

While we used the *Raidcomm V.2* in the local area environment and reduced the overhead in the system time by up to 70%, we can adopt *Raidcomm V.3* in a single-site configuration, which can reduce the overhead in the user time by up to 30%. One way to support the different kinds of underlying communication media and benefit from their characteristics is to integrate different communication models adopted in different versions of Raid communication subsystems under a unified interface. We are trying to build a new communication subsystem in our adaptable approach that can adjust itself based on the needs of the application.

5.1 Structural Details

The integrated communication facility takes into account the communication via various channels, including the communication through the main memory, through the local area network, and through the wide area networks (the Internet). It is based on the shared memory (among the processes and the kernel), an address-to-channel resolution mechanism, and a lazy data conversion strategy. We optimize the most common cases.

Processes communicate through channels. A channel is a logic abstraction of a lower level communication scheme. There can be more than one channel in the whole system. A message will be sent through one of them, based on the result of the (destination) address-to-channel resolution. There are three kinds of mechanisms used in Raid communication subsystems. *Raidcomm V.1* uses the general INET (Unix communication facility based on Internet protocol family) model, *Raidcomm V.3* adopts the SE protocol, and *Raidcomm V.3* employs the SM (shared memory local IPC) facilities. Each of them forms a channel in the integrated system.

In the SM scheme, the communication activity is local (inside the machine). To send a message through an SM channel, the sender writes the message in the shared memory segment and lets the kernel pass the control to the receiver process directly. In the SE scheme, the

data transmission involves the Ethernet. The sender puts the messages in a local port and invokes a system call to send the messages directly to the Ethernet using the SE protocol. The kernel on the receiving site gets the Ethernet packet, places the message in the destination port in the receiver host, and activates the receiver process. So the channel in this scheme can be considered as from one kernel to another kernel via the Ethernet. In the INET scheme, the messages can go to anywhere in the Internet. A special INET daemon is used to send and received the Internet datagrams carrying Raid messages. The communication between the local INET daemon and the sender/receiver process is through shared memory. When sending a message, the sender puts it in the port, and passes the control to the daemon. The daemon will send the message to its remote counterpart using the UDP/IP protocol. After the receipt of the message by the destination INET daemon, it will be placed into a port for the receiver to process. In this scheme the channel can be considered from one INET daemon to another INET daemon via the Internet.

5.2 Data Formatting

When high-level complex data objects are sent as messages, they are formatted only if the underlying communication medium cannot pass them directly. Different channels have different capabilities to transmit data in different data format. For example, SM is a two-dimensional random addressed storage just like the main memory. A complex data structure can be transmitted in SM without any conversion. To send a message through a SM channel, the sender process simply puts the data object there and passes the pointer to the receiver. The receiver gets the message as it is sent. However, data formatting is unavoidable in other channels. The SE channel is based on the low level ethernet packet, and the INET channel is based on the Internet datagram. Both of them can be viewed as one-dimension linear buffers. A complex data structure must be formatted first into the lower level buffers before transmitting, and must be converted back after it arrives in the receiving port. Both the SE channel and the INET channel use XDR. One more level LDG (long datagram) is placed in the INET channel below the XDR level (see section 2.1).

5.3 Address Resolution and Adaptability

The address resolution in the adaptable communication system is critical. The format of a Raid address has been given in section 1.2. There are two naming translation schemes. One is the Raid address to channel (and network address) translation, and the other is the Raid address to server (process id) translation. Both translations are performed with table look-up. The tables are stored in the shared memory.

Since Raid is an adaptable distributed system [BR89], it will restructure itself in response to failures or changing performance requirements. To support such adaptability in Raid, the new communication subsystem is reconfigurable. When adding, removing, or migrating a Raid server or a whole Raid site to meet the changes in the communication needs or the performance requirements, the communication subsystem can adjust itself.

6. Related Work

The requirements of a communication subsystem to support a distributed transaction processing system have been studied in [Spe86, Duc89, MB91a]. Many ongoing research paradigms that have contributed to this field are surveyed in [MB91a]. Efficient local and remote IPC has been investigated in various distributed computing systems such as V [Che88], Mach [Ras86], Amoeba [TvRvS+90], Sprite [OCD+88], *x*-kernel [PHOR90], Chorus [RAA+88] among many others. We have studied the performance implications of many of these ideas in the development of the various versions of the Raid system.

Local IPC. The interprocess communication confined inside one machine is a significant activity in the transaction processing of a local area network based or a wide area network based distributed system [MB91b]. Cross-address space communication has been proposed to increase system modularity, and extensibility. Kernel-level communication support has been discussed and evaluated in [Ber90]. It proposes a Lightweight Remote Procedure Call (LRPC) that is a kernel-based communication facility designed and optimized for the communication between address spaces on the same machine. It com-

bines the control transfer and the communication model of capability-based systems with the programming semantics and the large-grained protection model of RPC, to provide better performance [Ber90]. URPC (User-level Remote Procedure Call) moves thread management and the communication out of the kernel into each application's address space, eliminating the role of the kernel as an interprocess communication intermediary. It reduces the communication cost to 93 microseconds, which is nearly the lower bound [Ber90].

Remote communication. Recent development in distributed operating systems provides more efficient remote communication techniques for transaction processing. Many message-based operating systems provide the capability to send messages reliably to processes executing on any host in the network [MB91a]. Under the argument that the kernel should be simple to be efficient, many systems use the "micro-kernel" approach and implement many communication facilities outside kernel. Others believe that the high-level functionality can be pushed into the kernel to improve the performance at the cost of reduced flexibility. In the V system, a simple and fast message passing facility has been used as one of its fundamental building blocks. Interprocess communication in Mach is optimized for local case inside the same host by virtual memory techniques. Most of the key features in Amoeba are implemented as user processes; its remote communication is extremely efficient. In network operating system Sprite, the interprocess communication is implemented through the pseudo-device mechanism. RPC in x-kernel is implemented at the kernel level and high performance is achieved.

Communication protocol. Communication protocols address the important issues involved in the distributed processing of transactions, such as routing a message through the interconnected networks, and the reliable transmission of messages over unreliable networks. A range of protocols exists that vary in their reliability and efficiency. Cheap but unreliable datagram protocols such as IP are used to build up more reliable (and more expensive) protocols such as virtual circuit and the request-response protocols. Communication measurements are proposed and conducted in [PKL90].

The Versatile Message Transaction Protocol (VMTP) is a transport level protocol intended to support the intra-system model of distributed processing, where page-level file access, remote procedure calls, real-

time datagrams, and multicasting dominate the communication activities [Che86]. In order to support the 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 the 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.

The overhead of standard protocols cancels the high communication speed offered by modern local area network technology. Specialized communication protocols for LAN provide the opportunities for further optimization. Efficient streamlined protocols for high-speed bulk-data transfer and reliable multicasting schemes for the local area networks can optimize the resource utilization and reduce the communication delay. *Virtual protocols* and *layered protocols* have been used in the *x*-kernel to implement general-purpose yet efficient remote procedure call protocols [HPAO89]. Virtual protocols are demultiplexers that route the messages to appropriate lower-level protocols. For example, in the Internet environment, a virtual protocol will bypass the IP protocol for messages originating and ending in the same network. Layered protocols favor the reuse of the code through the mapping of a protocol's functional layers into the self-contained software modules. Communication protocols are implemented by assembling those modules in a given order. Similar ideas are also used in System V Unix streams [Rit84].

Broadcast and multicast. Broadcast and multicast are two important communication activities in the distributed database systems. Chang [Cha84] introduced a two-phase non-blocking commit protocol using an atomic broadcast. The support of the atomic broadcasting and the 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 the distributed data. These protocols demand expensive exchange of messages among data server processes. The broad-

casting capabilities of local area networks can be used to avoid unnecessary message exchanges [MSM89, KTHB89].

7. *Conclusions and Experiences*

Communication software is critical in the transaction processing systems. New applications require that up to more than one thousand transactions per second must be processed. In some financial institutions, this limit has already been exceeded. The response time for database transactions used in banks and air-line reservation systems are expected to be in the order of a hundred milliseconds. The speed of the CPU alone would not enable us to reach this objective. Communication time for round trips in the order of several milliseconds with little support for multicasting is not acceptable. In addition, the overhead due to extensive layering, generalizations, and the kernel involvement can be avoided to streamline IPC. We have identified several communication services and mechanisms that can make the system efficient. Separated address spaces can be used to structure a complex transaction processing system. However, such a structuring approach increases cross-address space communication activity in the system. When using the conventional kernel-based communication channels, the result is a communication-intensive system. Not only is the number of messages high but the messages are expensive to process. High interaction among servers also triggers costly context switching activity in the system, and the underlying operating system should provide special scheduling policy that reflects the high-level relationship among servers. The transfer of a complex data object needs sophisticated communication facilities that take into account the high-level demands. Increasing availability through the distribution and the replication of data demands specialized multicasting mechanisms.

We have shown that the main concern of communication in local networks is the performance of the local cross-address space communication mechanism. For a typical transaction, about 90% of the communication activity is local. In conventional IPC facilities that feature the remote/local transparency, the overhead due to the system calls and the context switch is expensive. We addressed this problem in the design of our new communication facilities. Our first prototype provides a streamlined interprocess communication service. It uses

mapped memory between the kernel and the server processes. This alleviates in part the demands imposed on the kernel. The use of our *Raidcomm V.2* in Raid results in a reduction of 60-70% of the system time. Context switching activity also diminishes because more messages can be processed during the same time slice. *Raidcomm V.2* has a straightforward mapping between server addresses and network addresses. It also exploits the semantics of transaction processing to provide an efficient multicasting support. Our second prototype combines the shared memory and direct control passing mechanism to support efficient interprocess communication. By eliminating XDR, and moving the communication and scheduling out of kernel, the use of our *Raidcomm V.3* in Raid results in a reduction of in the user time by 1/3. Adaptability can be achieved by integrating different schemes for different environments and changes in the performance requirements.

This research has been conducted in the Unix operating system environment. We believe that Unix does provide a good benchmark for experimental study of new ideas in an academic setting. We believe the communication technology in the hardware/media is advancing at a rapid pace. This research along with the related work in industry and academia contributes to the advances in the software.

In our current model of Raid, multicast addresses are added/deleted by Raid servers. The RC adds a new multicast address for a transaction, when it receives the first operation request for that transaction. In normal conditions, the AC deletes the multicast 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 the performance and the transparency in Raid.

Our work has been conducted in a local area network environment. 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 in 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 the transaction processing. A major objective should be the

reduction of the number of remote messages that are needed for the transaction processing. All these are questions that we plan to answer in the future.

Scheduling policies in conventional operating systems usually do not consider high level relationships that may exist among a group of processes. Although direct control passing has been used in our communication model to bypass the kernel scheduling, it was not a clean solution. We find the need to introduce the concept of a system of processes as a new operating system abstraction. In the new model, the scheduling can be done at two levels. At the higher level, the kernel would schedule systems of processes as atomic entities. Internally, the scheduling could be done based on internal requirements of each system. In particular, it could be based on its communication patterns.

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.

[Ber90] Brian N. Bershad. *High Performance Cross-Address Space Communication*. PhD thesis, University of Washington, June 1990. TR 90-06-02.

[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.

[BFHR90a] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, and John Riedl. Adaptability experiments in the Raid distributed database system. *In Proc of the 9th IEEE Symposium on Reliability in Distributed Systems,* Huntsville, Alabama, October 1990.

[BFHR90b] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, and John Riedl. Adaptability experiments in the RAID distributed database system. Technical Report CSD-TR-972, Purdue University, April 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.

[Bla90] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer,* 23(5):31-43, May 1990.

[BMR89] Bharat Bhargava, Enrique Mafla, and John Riedl. Experimental facility for kernel extensions to support distributed database systems. Technical Report CSD-TR-930, Purdue University, April 1989.

[BMR91] Bharat Bhargava, Enrique Mafla, and John Riedl. Communication in the Raid distributed database system. *Computer Networks and ISDN Systems,* pages 81-92, 1991.

[BR89] 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.

[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.

[Che88] David R. Cheriton. The V distributed system. *Communications of the ACM,* 31(3):314-333, March 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.

[DVB89] Prasun Dewan, Ashish Vikram, and Bharat Bhargava. Engineering the object-relation database model in O-Raid. In *Proc of the Third International Conference on Foundations of Data Organization and Algorithms,* pages 389-403, June 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 91-101, Litchfield Park, AZ, December 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.

[MB91a] Enrique Mafla and Bharat Bhargava. Communication facilities for distributed transaction processing systems. *IEEE Computer,* 24(8):61-66, August 1991.

[MB91b] Enrique Mafla and Bharat Bhargava. Implementation and performance of a communication facility for distributed transaction processing. In *Proc of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems,* pages 69-85, Atlanta, GA, March 1991. USENIX Association.

[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.

[OCD+88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer,* February 1988.

[PHOR90] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The *x*-kernel: A platform for accessing internet resources. *IEEE Computer,* 23(5):23-33, May 1990.

[PKL90] Calton Pu, Frederick Korz, and Robert C. Lehman. A methodology for measuring applications over the Internet. Technical Report CUCS-044-90, Columbia University, September 1990.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems,* 1(4):305-370, 1988.

[Ras86] Richard F. Rashid. Threads of a new system. *Unix Review,* 4(8):37-49, August 1986.

[Rit84] D. M. Ritchie. A stream input-output system. ATT Bell Laboratories Technical Journal, 63(8):1897-1910, October 1984.

[Ros89] Floyd E. Ross. An overview of FDDI: The fiber distributed data interface. *IEEE Journal on Selected Areas in Communications,* 7(7):1043-1051, September 1989.

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

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

[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.