1989

Communication in the Raid Distributed Database System

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

Enrique Mafla

John Riedl

Report Number:
89-892

COMMUNICATION IN THE RAID DISTRIBUTED DATABASE SYSTEM

Bharat Bhargava
Enrique Mafla
John Riedl

CSD-TR-892
July 1989

Communication in the Raid Distributed Database System *

Bharat Bhargava
Enrique Mafla
John Riedl

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

This paper identifies the basic functions required from a communication subsystem in order to support a distributed, reliable, reconfigurable, and replicated database processing environment. These functions include: reliable multicast, different types of remote procedure calls, inexpensive datagram services, and efficient local interprocess communication. We report on a series of experiments that measure the performance of several local interprocess communication methods, a kernel–level multicasting facility, the Raid system running on different network configurations, and a Push multicast program. Push is a tool that allows us to conduct measurements by supplementing and/or modifying the communication facilities in the operating system kernel while it is running.

*This research is supported by NASA and AIRMICS under grant number NAG-1-676, UNISYS, AT&T, and by a LASPAU fellowship.

1 Introduction

Many applications require connecting physically distributed database systems via a communication network. A single user transaction may need to access both local and remote data copies. The execution of a typical transaction in our distributed database system triggers eight local messages in each coordinating site and two rounds of remote message exchanges [BRb]. A distributed transaction running in a multiuser environment takes anywhere from 300 to 800 ms. On the other hand, the timing for a simple point–to–point exchange of messages range from 2–10ms. in a local area network (LAN) [BMR] to 500ms or more in a wide area network (WAN) [QH]. Communication times contribute in a major way to the processing time. Messages to manage detection of site failure/network partitioning, recovery, consistency, and replication, can further increase the time spent on communication. The main objective of this research is to build an efficient communication facility that can support a database environment featuring distribution, replication, reliability, and modularity [Bha]. An additional objective is to justify our designs and implementations through measurements in a distributed database environment.

1.1 Related Work

Until recently, remote terminal access and file transfer have been the main applications of computer communication [Che]. For these applications, point–to–point virtual circuits have provided adequate support. This paradigm has dominated the design and implementation of most existing transport protocols. The development of distributed applications demands new communication concepts such as remote procedure call (RPC), multicast, and real-time datagrams [Che]. Distributed database researchers have investigated and developed specialized communication subsystems to meet such needs [BRb,BJ].

The Versatile Message Transaction Protocol (VMTP) is a transport level protocol intended to support the intra–system model of distributed processing [Che]. It is optimized for page–level file access, remote procedure calls, real–time datagrams, and multicasts. 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). Multicast, datagram, and forwarding services are provided as variants of the message transaction concept.

In [Svo], Svobodova argues in favor of specialized communication protocols for LAN environments. The overhead of standard protocols cancels the communication speed offered by the modern LAN technology. Several experiments have shown that communication

in LANs is CPU intensive [BMR,LZCZ]. The elimination of unnecessary functionality allows the simple ethernet (SE) protocol to reduce the communication overhead in a LAN by fifty percent [BMR]. Application oriented protocols provide opportunities for further optimization. Efficient streamlined protocols for high–speed bulk–data transfer have been implemented and used in LANs [CLZ,Zwa].

The distributed shared virtual memory paradigm for distributed computing can provide high levels of transparency for interprocess communication and for memory management [LH]. The distributed database designer is presented with an abstraction of the network closer to a multiprocessor system than to a conventional point–to–point long haul network [PWP].

Specialized communication facilities have been used to improve the design and implementation of distributed database management systems. Chang [Cha] presents a two phase non-blocking commit protocol using atomic broadcast. The support of atomic broadcast and failure detection within the communication subsystem simplifies database protocols and optimizes the use of the network broadcast capabilities. Birman [BJ] uses a family of reliable multicast protocols to support the concept of *fault–tolerant process groups* in a distributed environment.

The functions that a communication subsystem should provide in order to support a distributed transaction processing system have been identified in the Camelot project [Spe]. RPC–based session services have been proposed to support the interaction among the data servers and applications. Besides the synchronous RPC having *at–most–once* semantics, other forms of RPC like asynchronous RPC or multicast RPC are also useful. Highly specialized datagram–based communication facilities can be used to satisfy the communication needs of the underlying operating system, on which the data servers and applications run. 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 the participants in a transaction to assist the transaction processing system at commit time.

1.2 Role of Communication in Distributed Transaction Processing

The response time of transaction processing is dominated by delays caused by interactions with external media, such as disks and networks. In single–site systems the critical issue is the performance of the disk I/O subsystem, and the interprocess communication subsystem. In distributed transaction processing systems, the critical issue is network communication. For instance, in a distributed system requiring 10ms per send/receive pair, a traditional two–phase commit involving ten sites would take 200ms. On the other hand, the same commit protocol implemented via a hardware multicast facility could decrease the commit time to 100ms. In the following paragraphs, we study the effect that communica-

tion delay, reliability of delivery, CPU cost of processing the messages, and expressiveness power of the communication primitives have on such distributed transaction processing issues like transparency to distribution and to reliability, communication topology of the transaction processing algorithms, computational model, and state detection.

Transparency to distribution/reliability is affected by the lack of powerful communication facilities like multicast and RPC [Che]. This forces the database implementor to provide her own communication services on top of the existing ones [PWP]. Reliable communication facilities can provide transparent support during failure and recovery [BJ]. Distributed shared memory can support high levels of transparency for the design and implementation of distributed data managers. Under this paradigm, the concepts of remote, local, primary, and secondary memories are unified by the distributed one–level storage abstraction [HT]. This abstraction can lead to simpler models of communication for the transaction processing system.

Decentralized protocols are symmetric and are easier to implement [Ske]. Better resiliency and higher levels of concurrency can be achieved with decentralized protocols by avoiding the bottleneck caused by a central coordinator. However, a decentralized protocol generates many messages when the communication subsystem does not provide suitable multicasting primitives, forcing the designers to opt for the centralized control. If instead of the point–to–point messages, the system provides kernel level multicast, a round of message exchanges for decentralized control can go down by up to 30 percent [BMRS].

Many researchers have employed the server–based approach to implement distributed transaction systems [BRb,LHM*,STP*,Lis]. This simplifies the design of the system and enhances its modularity, reliability, and reconfiguration capabilities; because every conceptual function in the system is implemented as a separate process with its own private address space. However, without efficient process management and interprocess communication this approach is not of practical use, at least in its original form. Several new ideas for efficient support of processes and IPC like threads, minimal kernels, lightweight RPC, etc, have been implemented in experimental systems [YTR*,DJA,BALL].

The complexity of distributed algorithms is due to the difficulty in establishing and agreeing in the system's global state by each of its components. This is achieved by sending control messages and receiving timely replies. Systems that do not guarantee small communication delays and variances lead to inefficient implementations of distributed algorithms. For example, timeout mechanisms are used for failure detection. The selection of the timeout interval depends on the communication delay and its variance. It must be short enough in order to be efficient, but at the same time, it must be large enough to avoid unnecessary retries.

In the next section, we identify the communication support required by the Raid distributed database system [BRb]. Section 3 contains a description of our measurements and experimental work that we have completed.

2 Communication Support for the Raid Model

Raid is a distributed database system specially designed for conducting scientific experiments [BNS,BLLR]. Besides transparent concurrency, atomicity, and data distribution, this system supports replication, recovery, and adaptability, together with the flexibility and modularity that is needed to support experimentation. We study transaction processing in the Raid system as an example of an operational system, and identify the basic features for the supporting communication subsystem.

2.1 The Raid Processing Model

Distributed transaction processing involves the following functions:

- Transparent access to distributed and replicated data.
- Concurrency control validation.
- Atomic commitment control.
- Recovery from failures (stable storage).
- Transparency to site/network failures (reconfiguration capabilities).

In Raid, each transaction processing function is implemented as a separate server. These servers can be arbitrarily distributed over the underlying network to form the logical database sites. However, in most cases, the group of servers that constitute a logical site reside together on a physical processing node. One way to implement such servers is as separate operating system processes. Operating system processes are a well understood unit of computation, and therefore can be used as a flexible, modular, and well–supported basis on which adaptability, reliability, distribution and their experimentation can take place.

Figure 1 depicts the organization of a site in Raid [BRb]. For each user in the system there is an action driver server (AD). AD executes the transaction which reads the data from the local database (currently, the database is fully replicated in each site). The other servers, i.e. atomicity controller (AC), concurrency controller (CC), access manager (AM), and replication controller (RC) exist on a per site basis. AC coordinates the global commitment/abortion of the transaction. CC enforces local serializability. AM controls access to the physical database. RC ensures consistency of replicated copies when some site is recovering from a failure. The servers interact with each other using the services provided by the naming and communication subsystems.

The schematic diagram in figure 2 shows a sample interaction among servers during a transaction execution. Other server–level communication topologies are possible. That

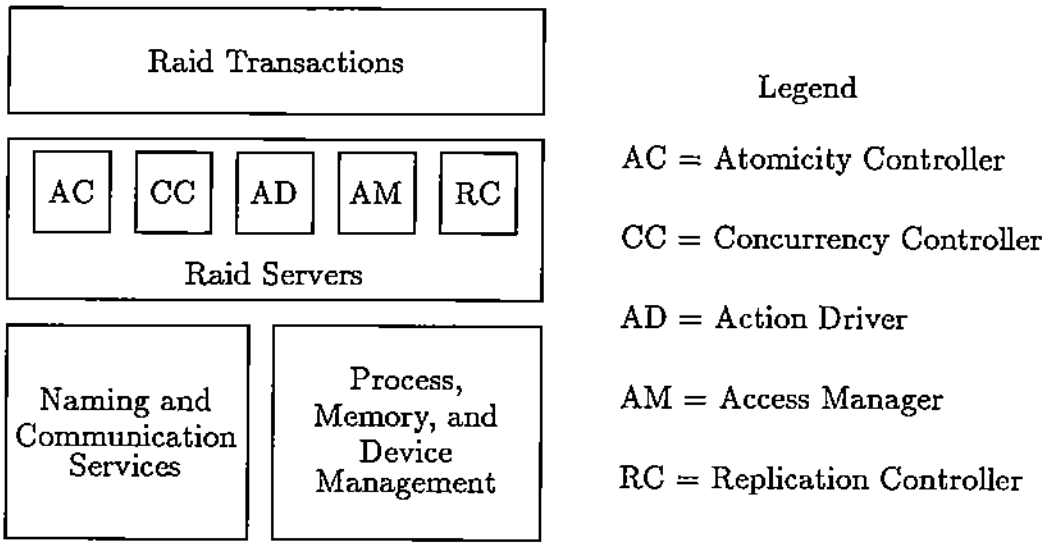
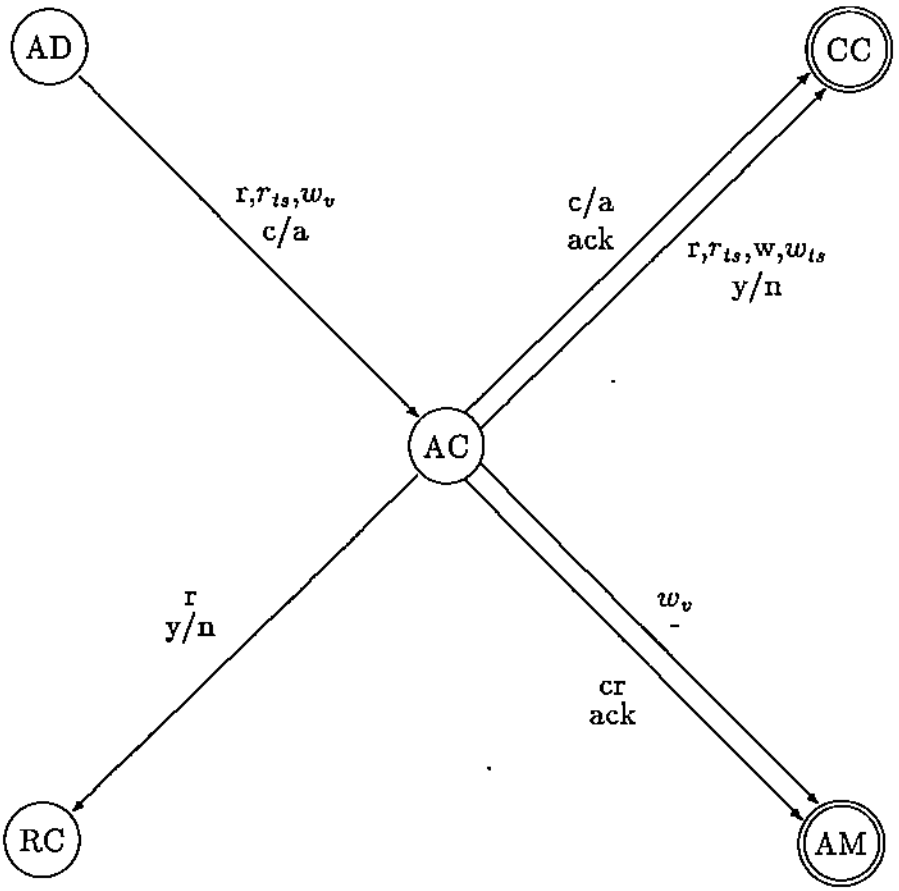


Figure 1: The organization of a site in Raid

interaction can be modeled as a sequence of remote procedure calls. A double circle represents the set of all servers (local and remote) of that type in the system. The arrow to such circles stands for a multicast remote procedure call. The nominator of the labels contains the parameters of the remote procedure invocation. The denominator is the information returned from the procedure call. After the AD executes the transaction locally, a commit protocol is started by the AC. The AC coordinates the commitment processing with all CC's and all AM's. It interacts with every CC and AM server (local or remote) in the same way. Conversely, the CC and AM servers do not have to differentiate between local and remote messages. This symmetry greatly simplifies the implementation of the servers. A typical execution flow accomplishing all these steps has been presented in [BRb]

2.2 Current Communication Support in Raid

In order to provide a clean, location independent interface between servers, Raid uses its own communication package. It is built on top of the Unix socket–based IPC mechanism. This package adds facilities for multicasting, arbitrary size messages, and a Raid–oriented naming service. Next we describe the Raid name space, the oracle (name server), and the available communication services.



Legend

r – read set	w_v – write values	y/n – yes/no vote
w – write set	r_{ts} – read timestamps	ack – acknowledgement
r_v – read values	w_{ts} – write timestamps	cr – commit record
c/a – commit/abort		

Figure 2: Transaction processing on a Raid site

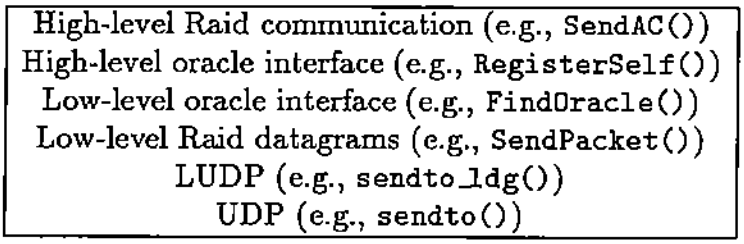


Figure 3: Layers of the Raid communication package.

The Name Space. We can uniquely identify each Raid server with the tuple (Raid instance number, Raid virtual site number, server type, server instance). To send a message to a server, UDP needs a (machine name, port number) pair. The Raid oracle maps between Raid 4-tuples and UDP addresses. The communication software at each server automatically caches the address of servers with which it must communicate. Thus, the oracle is only used at system start-up, when a server is moved, and during failure and recovery.

The Raid Oracle. An oracle is a server process listening on a well-known port for requests from other servers. The two major functions it provides are *lookup* and *registration.* A server can use the *lookup* service to determine the location of another server.

A server performs the `RegisterSelf()` call to permit other servers to locate it. RegisterSelf takes a single argument, called a *notifier set.* The notifier set is a list of regular expressions describing the Raid addresses of servers with which the new server must communicate. Whenever a server changes status (e.g., moves, fails, or recovers) the oracle sends a notifier message to all other servers that have specified the changing server in their notifier set. Notifier messages are handled automatically by the communication software, which caches the address of the new server. In many cases the higher-level code is not even aware of the reconfiguration.

The performance of the Raid Oracle only affects the start up and reconfiguration delays of Raid. The `RegisterSelf()`, `FindPartner()`, and `FindAll()` functions each require just a few packet round-trips. `FindOracle()` is an order of magnitude more expensive, since it must check for the oracle on all possible hosts on the network.

Raid Communication Facilities. The Raid servers communicate with each other using high-level operations such as `SendAC()`, which is used by the CC to send commit/abort messages to AC. Figure 3 shows the layering of the Raid communication package. LDG (Long Datagram) has the same syntax and semantics as UDP, but has no restriction on packet sizes. The fragment size is an important parameter to LDG. Normally we use

fragments of 8000 bytes, which is the largest possible on our Suns. Since IP gateways usually fragment messages into 512 byte packets we also have a version of LDG with 512 byte fragments. This allows us to compare kernel-level fragmentation in IP with user-level fragmentation in LDG.

Table 1 compares UDP, LDG, and Raid round-trip communication times for datagrams

Bytes:	64	512	2048	8192	32768	500000
UDP	7.2	10.6	16.5	48.8	-	-
LDG-512	10.2	17.2	41.9	147.4	550.0	8,630
LDG-8000	9.6	12.8	19.2	65.1	224.8	3,200
Raid	11.9	20.1	46.7	153.3	-	-

Table 1: Raid communication time by packet length (in ms)

of various lengths. The numbers given for the Raid layer are based on LDG-8000. This table shows the overhead introduced by the layers of the Raid communication package, which is more significant for larger packets. This overhead and the lack of a low–level multicast facility "serialize" the execution of the transaction across the nodes of the system, prolong the life of transaction, and consequently increase the probability of conflict with other transactions. The next subsection describes a set of kernel–resident communication facilities that can efficiently support the computational model of Raid.

2.3 Proposed Communication Support in Raid

We now present the main services of a kernel–level communication subsystem, oriented to support transaction processing in Raid. For environments that support a variety of applications, the benefits of communication standards are clear. However, for well defined applications running on dedicated LANs, the benefits are often outweighted by performance considerations. We can determine a priori the communication activity on the network. The communication subsystem can be optimized accordingly. In order to maintain the "open system" property, we can consider the whole LAN as one system, and provide a gateway to the external world [Svo].

Remote Procedure Calls. Interactions among the Raid servers can be readily modeled as remote procedure calls. The RPC paradigm facilitates the programming of the servers. Asynchronous remote procedure calls are needed to allow for concurrency. The types of remote procedures are known in advance. This permits to optimize their processing. The reliability of the RPC mechanism can be limited to the "at–most–once" semantics, because transactions can be aborted at any moment by any server [Spe].

With multicast RPC, the coordinating AC can reach all CC and AM servers at "almost" the same time. This increases the degree of parallelism in the execution of the transaction, which in turn reduces the amount of time the transaction spends in the system. Besides the corresponding improvement in response time, this parallelism guarantees the normal behavior of the database transaction processing system. Otherwise, the probability of aborting and blocking transactions will increase. Multicast RPC demanding only one reply or at least "n" replies can be useful to support alternate algorithms, e.g. quorum–based algorithms.

Datagrams. Datagrams can be used to support the secondary activities in the Raid model. Among the secondary activities, we have failure detection, clock synchronization, name resolution, simulation of experimentation events, etc. The processing of datagrams can be optimized significantly. The number and types of uses are limited and known in advance. The memory management to support Raid datagrams can be simplified accordingly. For example, we do not need the functionality provided by *mbufs* in Unix operating system. Fixed length buffers can be used instead.

Transaction Processing Algorithms. Because the communication subsystem has knowledge about the application, it can do more than just send and receive uninterpreted messages. With a little extra effort, the communication software can be of great help during the commit process or during update of replicated copies. For example, it seems reasonable to implement the entire commit protocol as part of the communication protocols [Spe]. As a result we can have not only a more efficient implementation of the commit protocol, but also a simpler implementation of the atomicity controller.

Naming Services. Raid needs specialized support for its name space. The naming services for the Raid distributed database system have to deal with failure, recovery, and adaptability events. By supporting the naming services inside the operating system, we can achieve a more efficient interaction between the naming and communication subsystems. This further increases the level of abstraction in the transaction processing protocols.

Flexible Resource Management. Finally, we have to provide an easy way to reassign the resources of the communication subsystem (e.g. network buffers, queues, shared memory, etc). This flexibility is needed during changes due to adaptability decisions, or experiment settings for example. We may need to dynamically change the number of network buffers assigned to the input queues of the servers that participate in remote communication. Or, we may want to experiment with alternative communication topologies of the Raid model. The communication subsystem has to be flexible enough to respond to those changes in the Raid communication topology.

3 Measurements and Experimental Data

In this section, we report the measurements on local communication, remote multicasting, various network configurations, and execution of Push programs. The Raid database system and SunOS 3.4 operating system are used in our measurements. Both systems run on Sun 3/50s connected to a 10 Mbps Ethernet[1] in our laboratory. To overcome the limitations in the clock resolution (20 ms in our case), we used the *ping* protocol [BMR]. A round-trip ping consists of a ping message sent from a designated ping process to a designated reflect process. The reflect process then returns the message to the ping process. Reported times are the average elapsed wall-clock time over several thousand round-trip pings. This method not only produces measurements with greater precision than the available hardware clock, but also amortizes start-up time over many messages. For some of the experiments, we use our own SE (simple ethernet) protocol [BMR]. This is a streamlined protocol implemented as a device driver, and was originally used to study the overhead of the UDP protocol [BMR].

3.1 Local Communication

There is significant local communication among various servers that implement a Raid site. In a five site Raid system, roughly half of the communication is local to one machine. Traditional IPC mechanisms are optimized for the remote case [BALL]. We investigate efficient methods for local IPC and for the integration of different IPC paradigms.

Design of the Experiment. We measure the performance of several local IPC methods available in SunOS. Two user–level processes: *ping* and *reflect* run the *ping* protocol described above. To see how each method scales with the message size, we use two types of messages, 10 and 1000 bytes long. The methods that we investigate are:

Two Message Queues - Message passing using two message queues; the first for messages sent from the ping process to the reflect process, and the second for reflected messages.

One Message Queue - Message passing using one message queue and two message types; the first message type for messages sent from the ping process to the reflect process, and the second for reflected messages.

Named Pipes - Two named pipes were created in the file system. One pipe was used for each direction of communication.

[1]SunOS and Sun are trademarks of Sun Microsystems, Incorporated. Ethernet is a trademark of Xerox Corporation.

METHOD	MESSAGE SIZE	
	10 Bytes	1000 Bytes
2 Q Message Passing	2.0	2.9
1 Q Message Passing	2.0	2.9
Named Pipes	2.3	3.9
Shared Memory	5.1	5.5
UDP Communication	4.3	9.6

Table 2: Local communication cost (in ms)

Shared Memory - Shared memory communication using two buffers and two semaphores for coordination. One semaphore was used for coordinating access to each buffer. The message was copied into the buffer but not out of the buffer, under the assumption that the message can be used in place.

UDP - UDP communication using unconnected packets.

Results. The results of the measurements conducted on these methods are displayed in Table 2. It demonstrates that there exist efficient alternatives to the socket–based method for local IPC. Queue message passing showed the least communication delay. Message passing using queues incurs 30 to 50 percent of the UDP delay, depending on the size of the message. Shared memory took substantially more time than message passing. Almost all of the elapsed time is due to the semaphore operations. The cost of UDP communication more than doubles as the message size increase from 10 bytes to 1000 bytes, while the cost of message queues and shared memory communication increases only by 45 and 8 percent respectively. This is because UDP messages are copied multiple times. Local UDP messages cost 2/3 as much as actually transmitting the packet on the network. That is why using a special mechanism for the local case is significantly faster. However, the integration of different IPC mechanisms for local and remote communication is not well supported. Unix uses the *select* facility for that integration. It introduces the overhead of an additional system call. For some cases, the *select* mechanism is not sufficient. For example, there is no means of allowing selection between message queues (or shared memory) and sockets.

Operating system policies such us memory and process management directly affect the performance of the IPC facilities. Context switching overhead in heavyweight processes can significantly increase the IPC times. The performance of the original implementation of Raid [BRa] was affected by such operating system based factors. The multi–server per process approach has been successfully used in Raid to avoid such high costs of local IPC and interprocess context switch [KLB]. 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 70 percent [KLB]. The multi-server implementation uses standard Unix facilities.

3.2 Remote Communication

Remote communication in the Raid system consists primarily of multicast messages. Multicasting is simulated in the high–level Raid communication package. This means that the CPU cost for message processing has to be paid several times in order to send the same message across various sites in the system. To alleviate this problem we have designed and implemented a multicasting facility inside the operating system kernel [BMRS]. This facility is implemented as a pseudo–device driver. It is both efficient and independent of the underlying network. The simulation of the multicasting process takes place at the lowest level possible just before enqueuing the packet into the network output queue. This mechanism can be implemented on networks that do not support physical multicasting.

Design of the Experiment. The *ping* process uses the *ioctl* system call to set the multicast address (which actually is a group of addresses) and the *write* system call to multicast the corresponding message to all destinations. The *reflect* processes have only one destination in the multicast address (that of the *ping* process). For this experiment, we use a variation of the *ping* protocol. The *ping* process does not wait for any reply before starting the next round of messages.

Results. The use of the kernel-level multicast facility results in savings of 40 percent when compared with the simulation of multicasting at the user–level (using the SE protocol in both cases). The overhead (0.7 ms.) per additional site is due to the implementation of the network device driver that we used. Our device driver does not handle efficiently back–to–back packets on the output queue (it takes approximately 0.45 ms to send a packet once it is enqueued into the output queue). A UDP–based implementation of the multicasting device driver would show further improvements. For instance, the multicasting of a message to ten destinations using this UDP–based kernel–level multicast facility would take 8.75 ms., while its user–level implementation takes 20 ms. In addition to the performance improvements, the use of the kernel–level multicasting primitive simplifies the implementation of the transaction processing algorithms.

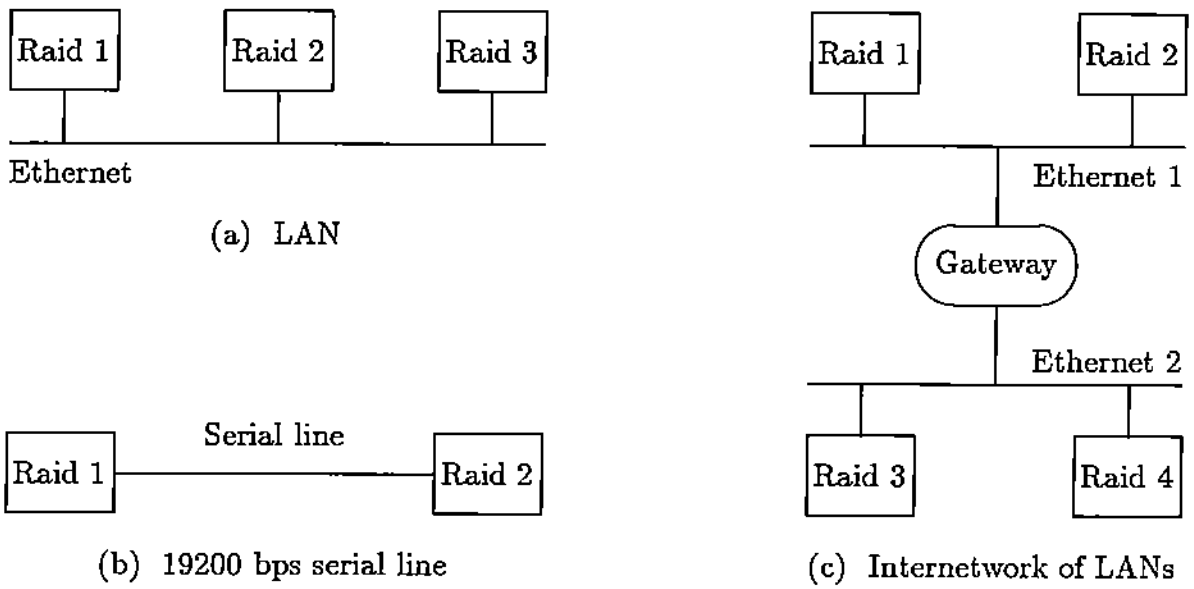


Figure 4: Network configurations for Raid

3.3 Raid on Different Network Configurations

We have studied the effects of the underlying communication network on the performance of the Raid system. We have to interleave computation and communication in a way that maximizes the degree of concurrency. Network technologies vary significantly in latency, throughput, and reliability. Therefore, the interleaving has to be adjusted for each specific network. The answers to database processing questions such as, "What distribution and/or replication policy should we use?" is different for different networks and different communication services.

Design of the Experiment. We run Raid on three network environments. First, we experiment with one–site and two–site Raid systems on a 10 Mbps Ethernet LAN (figure 4–a). For the second experiment, we use low bandwidth serial lines (figure 4–b). We are interested in this experiment because several wide area networks still use this type of links for communication. Furthermore, these lines can be used for backup purposes during link failures. Finally, we connect two 10 Mbps Ethernets with an Internet gateway. We collected data for a two-site Raid system, each site running on a different Ethernet (figure 4–c). The transaction benchmarks and measurements tools are as described in [BRb]. We time only the cost of committing the transaction; the cost of parsing the query is ignored. In the one–site case, all communication is within the same address space.

Transaction	LAN (1 site)	LAN (2 sites)	Serial line	Gateway
select one tuple	100	100	240	120
insert twenty tuples	320	380	520	400
update one tuple	100	120	260	120

Table 3: Transaction execution time (in ms).

Results. The results of these experiments are summarized in table 3. Message formatting and communication delay affect the execution time of transactions running on multiple–site Raid systems. Formatting cost depends on the size of the transaction. Communication delay includes communication protocol processing, and transmission delay by the underlying network hardware. Columns 2—4 in table 3 show the effect of these two factors. In a local area network, the difference between one and two sites is insignificant for small transactions. The clock resolution of the Sun workstations (20 ms.) almost blurs out this difference. However, for large transactions, the difference is noticeable. As reported in [BRb], round trip delays for messages using the Raid communication package in an Ethernet environment vary from 11.9 ms for 64–byte messages to 46.7 ms for 2048–byte messages. The transaction execution time is almost the same whether the two sites are running on the same Ethernet or on different Ethernets connected through a gateway. This result was expected, because the extra hop in the communication path introduces an overhead of about 5 ms, whose effect is lost in the clock resolution. The contrast between columns 2 and 4 reveals the impact of the slow serial line on the Raid performance. The transmission time for 1000 bits grows from 0.1 ms on the 10Mbps Ethernet cable to 52 ms. on the 19200 bps serial line. The observed difference (about 120 ms) is completely due to this transmission delay.

3.4 Experiments with the Push System

In conventional operating systems, the network facilities reside inside the kernel. The organization of kernel–resident services is not modular. For performance reasons, IPC is often layered on top of the network services [LMKQ]. Experimentation with new communication concepts in this environment becomes a cumbersome task. Changing the functionality of the network services demands expertise on large amounts of complex code and data structures. To solve this problem, we have developed a tool called *Push*. Push allows to load and execute user code inside the kernel, while the system is running. Appendix A presents design and implementation details of Push.

Number of destinations	kernel level SE	user level SE	Push
1	1.2	1.2	2.7
5	4.2	5.9	6.6
10	8.0	11.7	11.0
15	11.7	17.5	15.6
20	15.4	23.4	20.2

Table 4: Multicasting timing (in ms)

Design of the Experiment. We run the multicast Push program shown in appendix B. The parameters of this program are: a set of destination addresses, the number of those addresses, the message being sent, and the length of the message. The program loops taking one destination address at a time and sending the message to that destination. The *send* primitive is a "hook" that takes its parameters from the Push stack, transforms them into the Unix format, and invokes the Unix facilities to send the message. In particular, *send* formats the ethernet header, copies the message into mbufs, and enqueues the mbufs into the network interface output queue. To time the performance of this Push program, we use the modified version of the *ping* protocol (see 3.2).

Results. Table 4 compares the performance of the Push multicast program with the performance of the kernel–level and user-level SE multicast tools [BMR]. All three services provide the same functionality. Although the virtual machine still needs to be tuned, the results are encouraging. The execution of the loop in program B (12 Push instructions) takes about 300 μs, which averages 25 μs per instruction. This performance is comparable with that of the packet filter [MRA]. The relative high start–up cost (we measured 2.7ms for a single destination and 0.9ms per additional destination) can be optimized by reducing the number of times Push has to cross the user/kernel boundary.

4 Conclusions and Future Work

Our experiments show the performance improvements that can be gained from an adequate communication support. Optimized local IPC methods and multicast facilities can reduce transaction processing time significantly. For instance, the cost of message queues is approximately 1/3 the cost of local UDP communication. Therefore, in a five site Raid system, where local communication accounts for half of the total communication activity,

there is a potential savings of up to thirty percent for replacing UDP with message queues for local IPC. The use of a lower level multicast facility can significantly reduce the cost of remote communication, decreasing further the elapsed time for transaction processing. The use of Push simplifies the experimentation task. The implementation of the SE multicast device driver took 6 months. In contrast, writing the Push multicast program is a matter of minutes.

The first priority in the area of communication is the implementation of a communication subsystem, which can successfully respond to the requirements imposed by the Raid model. The insight gained with the SE protocol, the multicasting pseudo-device driver, the local IPC experiments, and the multi–server paradigm, together with other ideas proposed elsewhere [Spe,Svo,Che] will be used in design and implementation of the prototype.

We also want to develop further the Push approach to flexible operating systems. For instance, Push can be used to test/implement new *stream* modules. We want to tune the virtual machine to reach the best possible performance. This is very important to expand the use of Push as an operational tool. Second, we want to validate it as a reliable experimental tool. The main concern here is due to the interpretative nature of the system. We want to guarantee certain reliability about the knowledge that can be inferred from the Push experiments.

References

[BALL] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. *Lightweight Remote Procedure Call.* Technical Report 89–04–02, University of Washington, April 1989.

[Bha] Bharat Bhargava, editor. *Concurrency and reliability in distributed systems.* Van Nostrand and Reinhold, 1987.

[BJ] 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.

[BLLR] B. Bhargava, F. Lamma, P. Leu, and J. Riedl. *Three Experiments in Reliable Transaction Processing in RAID.* Technical Report CSD–TR–782, Purdue University, June 1988.

[BMR] 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.

[BMRS] Bharat Bhargava, Enrique Mafla, John Riedl, and Bradley Sauder. Implementation and measurements of an efficient communication facility for distributed database systems. In *Proc of the Fifth International Conference on Data Engineering*, Los Angeles, California, February 1989.

[BNS] Bharat Bhargava, Paul Noll, and Donna Sabo. An experimental analysis of replicated copy control during site failure and recovery. In *Proc of the 1988 Data Engineering Conference*, pages 82–91, Los Angeles, CA, February 1988.

[BRa] Bharat Bhargava and John Riedl. Implementation of RAID. In *Proc. of the 7th IEEE Symposium on Reliability in Distributed Systems*, Columbus, Ohio, October 1988.

[BRb] Bharat Bhargava and John Riedl. The Raid distributed database system. *IEEE Transactions on Software Engineering*, June 1989.

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

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

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

[DJA] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. *The Clouds Dsitributed Operating System.* Technical Report GIT–ICS–88/75, Georgia Tech, November 1988.

[HT] Meichun Hsu and Va–On Tam. *Managing Databases in Distributed Virtual Memory.* Technical Report TR-07-88, Harvard University, March 1988.

[KLB] Charles Koelbel, Fady Lamaa, and Bharat Bhargava. Efficient implementation of modularity in Raid. To appear in the USENIX Workshop on Experiences with Building Distributed (and Multiprocessor) Systems, oct. 1989, Florida.

[LH] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proc of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[LHM*] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[Lis] Barbara Liskov. Distributed programming in ARGUS. *Communications of the ACM*, 31(3):300–312, March 1988.

[LMKQ] 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.

[LZCZ] 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.

[MRA] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[PWP] Thomas W. Page, Mathew J. Winstein, and Gerald J. Popek. Genesis: A distributed database operating system. In *Proc of ACM-SIGMOD 1985 International Conference on Management of Data*, pages 374–387, May 1985.

[QH] John S. Quarterman and Josiah C. Hoskins. Notable computer networks. *Communications of the ACM*, 29(10):932–971, October 1986.

[Ske] D. Skeen. A decentralized termination protocol. In *Proc. 1st IEEE Symp. on Reliability in Distributed Software and Database Systems*, pages 27–32, Pittsburgh, PA, July 1981.

[Spe] Alfred Z. Spector. *Communication Support in Operating Systems for Distributed Transactions.* Technical Report CMU-CS-86-165, Department of Computer Sciences, Carnegie Mellon University, November 1986.

[STP*] 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.

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

[YTR*] 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 *SOSP87*, pages 63–76, ACM, Austin, TX, November 1987.

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

APPENDICES

A Design and Implementation of Push

The Push approach is similar to the packet filter described in [MRA], in which user specified code can be dynamically loaded into the kernel to demultiplex packets for user-level implementations of network protocols. The difference is that the packet filter simply specifies destination processes for the packets, whereas our routines would be able to collect multiple messages, generate response packets, and only return to the user once for a complex interaction. For instance, a multi-phase commit protocol could be written in this language that would send and receive two rounds of messages with a single system call.

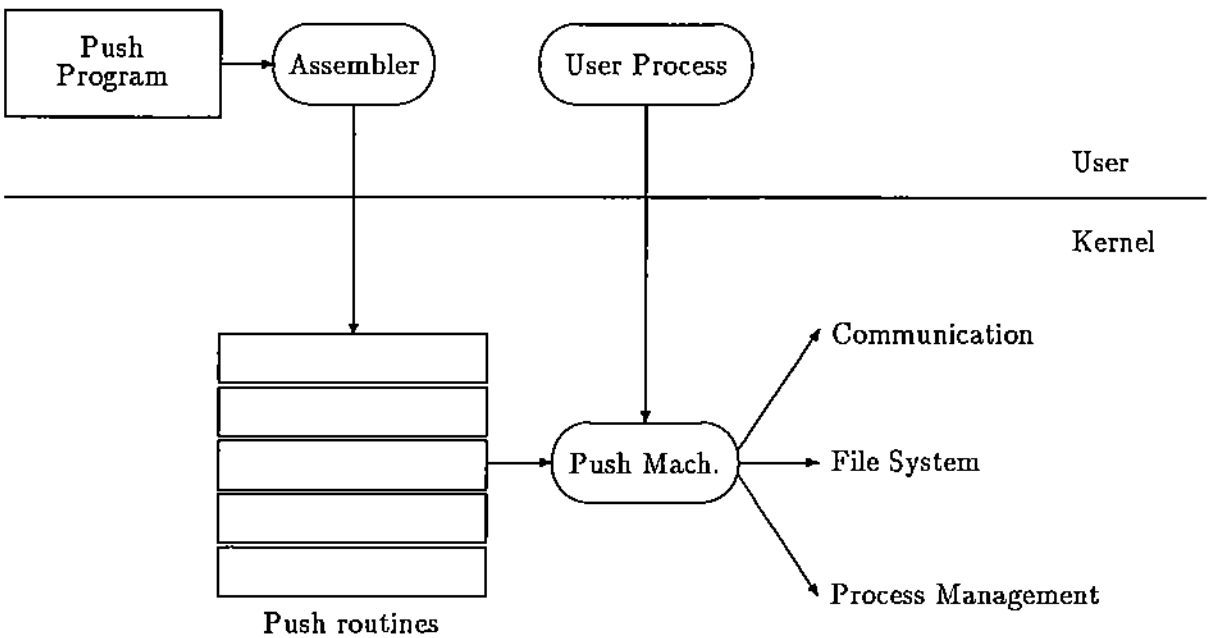


Figure 5: The Push system

The Push system provides a KUP (Kernel–level execution of User–level Protocols) language. The user can specify complex communication functions in this language. The core of the system is a virtual machine, able to run assembled Push programs inside the kernel. Figure 5 shows the details of our idea. The user's Push program is first assembled and then added to the kernel. Once this is done, the user can invoke the new functionality implemented by his/her Push program.

The Push language is a simple, general-purpose stack–based language. Each Push program contains a declaration of the input/output parameters, a definition of local variables and the code itself. There are two data types in the language: integers and addresses (pointers). The code section is a sequence of Push operations. Each operation can have one argument and a label which can be referred in jump instructions. The program in appendix B illustrates the features of this language.

The assembler translates the user–level programs to an internal representation, expedient to be executed by the virtual machine. These assembled programs are then stored inside the kernel in special data structures for later execution.

The virtual machine executes Push programs on behalf of the users. The interpreter first checks the parameters and copies them to the Push space. To protect the kernel space, each access to the Push space is strictly controlled. The program can allocate/free additional memory from its Push space. During execution time, each Push process is provided with an execution stack, which contains the parameters, local variables, and the values dynamically pushed into it while the program is running.

Besides the basic stack operations, the user is provided with primitives that allow him/her to access existing networking services. For example, send and receive operations permit the user to access different levels of the communication protocol hierarchy, e.g. network device drivers, internet layer, etc. With these "hooks" to the rest of the operating system, we can make use of the functionality already existing inside the operating system. Currently, we have a prototype of the Push system which provides facilities for communication experiments.

B A Sample Push program

```
%Push multicast procedure

addrlen   def     6

addrs     in      address
addrcnt   in      integer
msg       in      address
msglen    in      integer

nxtaddr   var     address

          push    addrs     % initialize for looping through addrs
          pop     nxtaddr

loop      push    nxtaddr   % deliver message
          push    msglen
          push    msg
          send

          push    nxtaddr   % compute addr of next destination addr
          push    addrlen
          add
          pop     nxtaddr

          push    addrcnt   % decrement count of destinations
          dec
          dup
          pop     addrcnt

          jgt     loop      % continue if more destinations

          return
```