Implementation and Measurements of Efficient Communication Facilities for Distributed Database Systems

Bharat Bhargava, Enrique Mafla, John Riedl, and Bradley Sauder
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract
This research presents our experimentation with several methods of providing efficient communication facilities for distributed database systems. These studies give insight into the delays incurred by applications running on distributed systems. We have implemented, compared, and analyzed five different mechanisms for local interprocess communication (two variations with message queues, named pipes, shared memory, and UDP sockets). The most efficient of these is three times as fast as UDP for 1000 byte messages. We implemented and measured the performance of kernel-level software multicast and hardware multicast. The results show the significant advantage of using these techniques instead of using multiple sends and receives at the user level. Finally, we present the design of a facility that allows the dynamic addition of user-level protocols such as two phase commit, clock synchronization, etc. to an operating system kernel. The facility is based on a simple stack-based language that provides the functionality and security required. This tool makes available to the user the best of two worlds. It provides the efficiency of kernel-resident code with the flexibility and safety of user-level programming.

1 Introduction
Our motivation for experimenting with communication facilities is the improvement of transaction processing performance in a distributed environment. The performance and general behavior of distributed applications depend on the communication system used by those applications. The communication services supported by conventional operating systems have proven to be inadequate [5,12]. We are working with the Raid system [3] to determine the bottlenecks due to communication delays in distributed database systems, and to test the new communication facilities that we have developed.

1.1 Communications in Transaction Systems
In order to improve the performance of the communication system we need to consider its characteristic usage. During transaction processing the communication consists of relatively small request-response messages, most often between servers on the same site. Occasionally these messages will be directed to servers on remote sites to request data not available locally. During transaction commitment small control messages will be exchanged between the home site for the transaction and all sites with which the transaction communicated. In Argus [6] and Camelot [10] messages are sent to each data server involved in the transaction, while in Raid a single server on each site, the Atomicity Controller, processes the messages. Two complete rounds of these control messages need to be exchanged. Finally, updated copies of modified data items are distributed to replicated copies as necessary. This data will sometimes involve considerably larger messages when a transaction makes many modifications to data items. Note that this analysis ignores the events during transaction processing that do not ordinarily generate messages. In Camelot there will potentially be additional messages as log messages are to remote or replicated log servers.

We now discuss two characteristics of this message traffic. First, the message sizes show a bimodal distribution, with most messages being small control messages, interspersed with occasional bursts of the larger messages containing updates to data items. This is the same distribution seen in experimental studies of actual Ethernet traffic in other environ-
The larger messages are usually directed to remote hosts, since replication is more useful between different hosts\(^1\). Second, the messages are divided between single-destination messages used during transaction processing and multiple-destination messages used during commitment. Judicious use of low-level multicast support can offer significant performance improvements. Maintenance of multicasting groups is a significant overhead of multicasting protocols in many environments. Since similar transactions are likely to use similar sets of servers, this overhead should be low for transaction systems.

In essence, local IPC, multicasting, and flexibility and efficiency in the implementation of distributed protocols are some of the important issues that should be addressed when dealing with the role of communications in distributed transaction systems. These are precisely the problems that we investigated in our experiments. Sections 2 and 3 explore methods for improving communication performance. Section 2 describes our measurements and analysis of the performance of five different methods for local communications, and includes a description of techniques for integrating special-purpose local communications methods with a remote communications system. In this way the overhead of the general-purpose protocol is avoided for an important special case. Section 3 explains our kernel-level software multicast and our hardware multicast implementations, and gives the results of our experiments with multicasting. Most of the remote communication in Raid is directed to all of the sites in a Raid system. Sending all of these messages in a single system call can dramatically reduce communication cost. Section 4 introduces Push, a new system allowing users to dynamically implement kernel-level services. Push provides the performance of kernel-level protocols, but insulates the user from many of the problems involved in kernel programming. Finally, Section 5 gives the conclusions we have drawn from our experimental work, and suggests future research directions.

### 1.2 Measurement Tool for Experimentation

The measurements presented in this paper were conducted using the ping protocol described in [2]. All experiments were run on an Ethernet with SUN workstations using the Unix\(^2\) operating system. Here we briefly describe the experimental method and mention some previous results that have impact on the studies presented in this paper.

The basic method used to collect these results is the ping protocol. 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 for several thousand round-trip pings. This method amortizes start-up time over many messages, and can produce measurements with greater precision than the available clock hardware. Components of the UDP protocol were measured by developing our own SE (simple ethernet) suite of protocols [2]. Each of the SE protocols is capable of transmitting and receiving messages, but each leaves out some part of the UDP/IP processing. For instance, one version is implemented as a device-driver rather than a socket, another does not compute checksums, etc. The SE protocols provide a method for indirectly measuring the costs of the various components.

About \(\frac{1}{3}\) of the time spent in delivering UDP/IP messages is spent getting into and out of the kernel. The socket layering abstraction is \(\frac{1}{3}\) of the total, mostly because of the large number of procedure calls used to support it. The most significant data manipulation is in the mbuf\(^3\) allocation and deallocation, and the kernel space to user space copy, and together these are only about half as expensive as the socket layering. The lesson from this study is that decreasing the time needed to invoke a kernel service, or decreasing the number of times kernel services are invoked are effective ways to improve datagram speed. Furthermore, this result suggests that other methods for improving communication performance must take into account the overhead of kernel invocation. For instance, some proposals suggest that a separate communications processor could be used to decrease datagram latency. However, our study shows that most of the expense of the communication will still be borne by the primary processor, unless a way is found to communicate with the communications processor other than through a kernel call.

### 2 Local IPC Performance Improvements

One problem with using Unix for building database systems is the poor performance of the Inter-Process Communication (IPC) suite of protocols. The ping protocol is a trademark of AT&T Bell Laboratories. The mbufs are the buffers in which network data is passed around in the kernel.
Communication (IPC) mechanisms [12]. UDP is an expensive form of communication because of the number of layers of software involved in the UDP sends and receives [2]. During protocol processing, UDP copies the message multiple times. This section describes an investigation of alternatives to the User Datagram Protocol (UDP) for local IPC in the RAID system. The best of these alternatives are nearly optimal, in that each message is delivered in little more than the time required to copy the message twice.

The IPC alternatives investigated include the following communication mechanisms available on Sun Unix version 3.4:

**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.

**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 sockets.

The results of these tests are shown in Table 1.

<table>
<thead>
<tr>
<th>METHOD</th>
<th>MESSAGE SIZE</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 Bytes</td>
<td>1000 Bytes</td>
<td></td>
</tr>
<tr>
<td>2 Q Message Passing</td>
<td>2.0</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>1 Q Message Passing</td>
<td>2.0</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>Named Pipes</td>
<td>2.3</td>
<td>3.9</td>
<td></td>
</tr>
<tr>
<td>Shared Memory</td>
<td>5.1</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td>UDP Communication</td>
<td>4.3</td>
<td>9.6</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Local Communications Costs (in ms)

2.1 Analysis of Results

Message passing showed the least communication delay and UDP communication showed the highest delay in all cases. Message passing using queues incurs \( \frac{1}{2} \) to \( \frac{1}{3} \) the delay of UDP, depending on the size of the message. Shared memory with semaphores took substantially more time than message passing. This result was especially surprising since the shared memory approach only copied the data into the shared segment and not out of it, while the message passing implementation copies data both ways. Omitting the copy from the 10 byte shared message experiment did not change the result, indicating that almost all of the elapsed time is due to the semaphore operations. The high cost of semaphores is probably due to the complicated semaphore semantics. In particular, simple semaphores can be implemented more cheaply as message queues with one byte messages! The semaphore package provides an undo capability for cleaning up when a process exits. The numbers presented here do not use this facility, which costs an additional 0.3 ms per round-trip.

The semaphore experiment took the same amount of time even if the copy operation just moved data into a global data area in each process, showing that shared memory has the same speed as local memory. The difference between the 10 byte and 1000 byte versions of the shared memory experiment should be the time to copy 1000 bytes twice (once from the ping process to the shared memory segment and once from the reflect process to the shared memory segment); 0.4 ms seems a reasonable figure for a Sun 3/50. Queue message passing increased the cost by almost exactly twice the copying cost when the number of bytes was increased. This is optimal for communicating between disjoint address spaces. Two queue message passing does slightly better (0.02 ms) than one queue message passing, probably because one queue message passing incurs some overhead in manipulating message types.

The cost of UDP communication almost triples as the message size increases from 10 bytes to 1000 bytes. Part of the reason the cost of UDP communication increases so rapidly with the size of the packets is that the message is copied multiple times. The special-purpose methods for local communication use simple protocols that generally do not require any extra copying of data. Note that the Sun implementation of UDP checks for the case of a message with a destination on the local host in the UDP layer. This check is after the socket layers, and just after the data is copied into mbufs, but comes before any part of the IP protocol. Including the IP protocol would increase
the cost of UDP by an additional $\frac{1}{6}$.

In a five-site RAID system, roughly half of the communication is local to one machine. The cost of local communication using message queues is approximately $\frac{1}{2}$ the cost of local UDP communication. Therefore there is a potential savings of up to thirty percent for replacing UDP with message queues for local IPC. Using a lower level multicast facility (see Section 3) would significantly reduce the cost of remote communications so that an even larger percentage of the total communication cost would be from local communication. In a five-site RAID system using hardware multicast, about $\frac{1}{2}$ of the communication is local. Reducing the local communication cost by a factor of three in this system would reduce total communication cost by almost $\frac{1}{2}$.

2.2 Conclusions

The results of the timing experiments show that all of the local IPC mechanisms tested are more efficient than UDP. In particular, message queues can provide a 50 percent improvement in performance for short messages (60 percent and more for larger messages). However, message queues can be used only for local communications. A method of integrating use of an efficient local IPC mechanism with a more general mechanism for inter-site communications is required. The `select()` system call provides this facility for IPC mechanisms that are implemented as devices (e.g. named pipes and UDP sockets) but will not work for semaphores or message queues. Integrating message queues with UDP will require either building a device driver for message queues or developing a new method of waiting for a message from two different places. Another problem is that `select()` just returns an indication of which device has a message. A separate system call must be used to actually read the data. We performed additional measurements that show that the cost of the existing `select()` call is approximately 1.5 milliseconds per message round trip. This means that an overhead cost of 0.7 milliseconds must be paid for every message in a system that uses `select()`. Waiting on multiple devices would have little additional overhead if the data were returned on the same call. The next step in our research is to implement a mechanism that will wait on the receipt of messages at a message queue or a UDP socket and then return that message with one system call. This mechanism will then be tested in the RAID system.

3 Multicast Communication Experiment

Multicasting of the same message to different components of the system is required by many distributed protocols. Currently, many distributed systems are implemented on top of conventional operating systems, which usually provide support only for point to point communications. Those systems, when faced with the necessity for multicasting, resort to its simulation at the application level, which results in poor performance. The experiments described in this section deal with the implementation of multicast facilities at the kernel level. The goal is to contrast the performance of this approach with the corresponding implementations of the same service at the user level.

The most efficient way to implement multicasting in a broadcast-based environment like Ethernet is through the use of the built-in capabilities of the network. In Ethernet, the cost of sending a message does not depend on the number of destinations. We implemented hardware multicasting by modifying the Lance Ethernet device driver. Three changes were necessary. First, we created a new control call to download to the Lance interface the multicast addresses to be received by a host. Second, we modified the driver to recognize multicast packets and return them to the user. Finally, we modified our SE protocol to support the transmission of multicast packets.

However, hardware multicasting is usually not available to the user in conventional operating systems. Furthermore, many networks do not support hardware multicasting at all. An attractive alternative to hardware multicasting is the simulation of its functionality in the kernel. With this approach, a single system call causes the kernel to transmit separate messages to each of the specified destinations. Given the results of the experiments in [2], we expect this solution to scale well as the number of destinations increases. To this purpose, we implemented a device driver to simulate multicasting. The driver maintains status information in a data structure known as a `control block`. The control block for this device includes the addresses of the current destinations. Before starting any packet transmission, the user must set these addresses using a control function. We assume that multicasting activity in the system usually involves the same set of destinations, so the control function will not be invoked often.

To conduct the experiments, we used one ping process, which repeatedly sends messages to multiple destinations according to three different approaches. These approaches for multicasting are kernel-level simulated multicast, physical multicast, and user-
level simulated multicast. Before performing the experiments, we tested the code with the complete ping-reflect mechanism described in Section 1. The results of the experiments are summarized in Table 2.

The results indicate approximately a 38 percent improvement in performance of the kernel-level simulated multicast facility when compared with the user-level simulated multicast. In both cases the time increases linearly with the number of destinations. However, the extra cost per destination for user-level multicast is 1.2 ms instead of the 0.75 ms that we have for the kernel-level case. Note that both the user-level and kernel-level multicast simulations use our SE protocol as a starting point. SE is already a stripped down protocol, performing round trips twice as fast as UDP. Performing the experiment with UDP would add an addition 1.5 ms per destination for the user-level multicast, while adding only a constant 1.5 ms to the kernel-level multicast. Sending to ten destinations, the user-level multicast would take \(2.0 \times 10 = 20.0\) ms, while the kernel-level multicast would take \(2.0 + 0.75 \times 9 = 8.75\) ms, for a savings of over 50%. As expected, physical multicast time does not depend on the number of destinations.

### Table 2: Timing for varying number of destinations.

<table>
<thead>
<tr>
<th>Number of destinations</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel level</td>
</tr>
<tr>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>5</td>
<td>4.2</td>
</tr>
<tr>
<td>10</td>
<td>8.0</td>
</tr>
<tr>
<td>15</td>
<td>11.7</td>
</tr>
<tr>
<td>20</td>
<td>15.4</td>
</tr>
</tbody>
</table>

3.1 Conclusion of the Multicast Experiment

Even though we did not put much emphasis on code optimization, the results obtained confirm our expectation that implementing multicasting at the kernel level will significantly improve the performance over a user-level implementation. More careful implementations of the same device driver or, even better, implementations of direct system calls will provide better performance results. A more significant performance improvement could be obtained by decreasing the incremental cost of sending to each site. For example, the multicasting can be done at a lower level, just before the packet is handed to the physical device.

For systems like RAID, it would be interesting to see how these new ideas of implementing multicasting affect performance. Furthermore, it seems that some distributed protocols, like the commit protocol, can be implemented as a whole at lower levels than the current application level. The idea here is to have a site invoke the commit protocol, run the commit protocol at the kernel level, and only have the final decision (commit/abort) be returned to the application level. The design of a mechanism to support kernel execution of user protocols is described in the next section.

4 Kernel-level Execution of User-defined Protocols

There has been much discussion on the implementation of transaction processing protocols in the operating system kernel [1]. The difficulty is that often services built into an operating system are not exactly the services needed by the implementors of distributed database systems. On the other hand, user-level implementations of these services often provide poorer performance than kernel-resident versions. This section discusses a general-purpose mechanism for implementing needed operating system services in the kernel in a safe and efficient manner. The mechanism is described in terms of network services, although it is applicable to many other operating systems services, such as process scheduling, memory management and file system buffer management.

We are investigating methods for dynamically loading special procedures such as a multi-phase commit protocol or a global clock synchronization algorithm into the kernel to process complete communication protocols with a single system call. These procedures will be written in a simple language that can be interpreted by the kernel in an efficient but safe manner. The approach is similar to the packet filter described in [7], 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 messages for

---

4 Currently we transmit multicast packets at the lowest device-independent level. Lower level multicast will have to be device-dependent.
4.1 Design Issues

A KUP (Kernel-level execution of User-level Protocols) language is a language with which a user can specify complex service requests that will be executed entirely within the kernel. KUP programs are invoked with special system calls. They can issue read and write requests using existing network services such as UDP, IP, or SE (our own protocol for raw ethernet access [2]). In designing a KUP language there are several conflicting goals:

1. A KUP language should protect the rest of the kernel address space from access by these programs. An erroneous program may produce incorrect results for its users, but it must not violate the integrity of the kernel.

2. Programs in KUP languages must be efficient to execute. If a KUP language is inherently slow, the primary goal of achieving very high performance cannot be met.

3. KUP languages should provide simple timer services to the programs. Services built-in to an operating system are most often deficient in handling boundary conditions, such as errors. In a distributed environment, error handling must include support for detecting lost messages.

4. A KUP program must not be able to monopolize the CPU.

There are several approaches to protecting the kernel address space from arbitrary access by KUP programs. The first is to develop a user-level compiler that produces type-safe code, compiling in run-time checks where necessary. The compiler would mark the programs in an unforgeable manner and a privileged loader would be the only program with permission to push programs into the kernel5. Alternatively, the kernel could accept programs in the high-level language and compile the programs itself. A difficulty with these two approaches is that the compiler would be difficult to port to new architectures. Also, loading compiled programs safely into the kernel would be tricky. Implementing a compiler in the kernel has the further disadvantage that it would increase the kernel size significantly, which would adversely affect performance.

Instead, we chose to design a virtual machine within the kernel for running KUP programs. We call the KUP language we are implementing Push. The Push machine is stack-based, with a simple instruction set. The first Push implementation is an assembler for this virtual machine that just provides mnemonic instructions and symbolic labels. After the virtual machine has stabilized, a future implementation will include a compiler from a subset of C to the assembly language. The language design provides for simple implementation. The assembler is 884 lines of C code, and compiles to 60 Kbytes, unoptimized. A Push disassembler is 332 lines of C code, and 20 Kbytes compiled. The interpreter is about half-finished, and so far consists of 233 lines of C code. The size of the interpreter is important, since it must be kernel-resident. We predict that the interpreter will take less than 10 Kbytes of memory. Ten Push programs of 100 statements each would consume about 5 Kbytes, including stack space. Our goal is for an entire Push implementation to increase the kernel size by less than 20 Kbytes. More details of the Push language and implementation are given in [8].

Performance is a potential problem of the virtual machine approach. The virtual machine instruction set is similar to the stack language in [7] which requires about 30 microseconds per interpreted instruction on a Microvax II. This execution time translates into an instruction rate of 30 instructions per millisecond. Receiving or sending a packet from within the kernel takes about 500 microseconds. Sample Push programs generally require about 5 instructions to transmit a packet, and slightly more to receive a packet. This will add about 30% overhead over a native machine code implementation, but should still be significantly faster than user-level implementations.

In addition to protecting the kernel address space, we must prevent processes running Push programs from monopolizing the CPU. The most important part of this protection is running the programs with interrupts enabled. While executing kernel routines such as 'receive', interrupts will be disabled as usual, but Push has no command to affect the interrupt status. Hence clock interrupts will occur as usual, and the kernel will make its normal time-slicing decisions. Unfortunately, Unix only replaces the executing process upon entering or exiting the kernel, and Push programs may loop indefinitely within the kernel. Our solution is to add code that checks for runaway Push programs to the clock interrupt routine. If a Push program is running when a clock interrupt occurs, the routine increments a special 'wound' counter in the Push program. If the wound counter is incremented beyond a fixed limit, the interrupt routine terminates the Push program, returning an error message to the user. In addition, the Push program

5 For instance, the compiler could include a cryptographic checksum in the compiled program.
is purged from the table of programs and a message is printed on the console, so that the same program does not continue to monopolize the CPU. Long-running Push programs may need a method to increase the number of clock ticks permitted.

Many of the Push programs will need timer services so messages can be retransmitted or timeout failures can be returned to the user. Our design supports a simple timeout facility that invokes the program at a specified label after a certain time (specified in milliseconds) elapses. The timeout is supported by the clock interrupt routine which keeps a list of pending timeouts in increasing order of time. When a timeout expires, the clock routine checks to see if the program is still active. If so, the clock routine cleans up any queues the program was waiting on, sets its execution point within the interpreter to the specified address, and returns the calling process to the run queue. When the process is rescheduled, it begins interpreting again.

4.2 Conclusions and Analysis

The potential performance improvements from using the Push system can be estimated based on the communications software measurements in [2]. For instance, consider the performance of a MultiRPC call to twenty sites. Using UDP, each send/receive pair will take 7.2 ms, 6.5ms of which is CPU time on the local host. Assuming that UDP can buffer all twenty replies, the sends and receives can be overlapped so that the total time will be about 6.5ms * 20 = 130ms. Alternatively, we can send and receive the same messages in Push with a single system call. A send/receive pair in Push will avoid the overhead of performing the system call and copying the data between user space and kernel space for each call, which total to about 30% of the cost of UDP messages. So the send/receive pair will take 5ms, 4.3ms of which is CPU time on the local host. Interpreting the Push program adds an overhead of about 30µs per instruction, for an additional cost of no more than 0.3ms per send or receive. In addition, there must be one entry to and one exit from the kernel at a total cost of 2.2ms. Thus the total time for 20 send/receive pairs in Push will be

\[
4.3ms + 20 + 0.3ms + 20 + 2.2ms = 94.2ms
\]

for a total savings of 28% per MultiRPC call.

Another potential use of the Push language is for applications that require guaranteed response times. For instance, many fault-tolerance protocols require the detection of failed hosts or networks within a bounded time. Also, real-time applications require that responses arrive before a specified time. Push can be extended with support for programs that will execute in lower level device drivers. The simplest example is the kping protocol described in [2]. kping is a kernel protocol that responds to ping messages in the lower level device driver for the Lance ethernet interface. From the time kping is invoked by a kernel to the time the reply is received is 1.3ms. By contrast, ICMP, the Arpanet ping protocol takes 5ms for a round-trip ping. Furthermore, ICMP is very sensitive to the load on the target machine, which makes it difficult to use for detecting failures. Algorithms to synchronize global clocks are another application that could make use of the Push service. The clock could be maintained in the kernel using a protocol written entirely in Push. The protocol would be initiated at system startup, and would then synchronize the clock with other computers without ever returning to user level. Note that this application requires special consideration from the clock interrupt routine that detects runaway Push programs, so that it is not terminated for running "too long".

5 Conclusions and Future Work

Our measurements of enhanced communications support for transaction processing in distributed systems offer insight into methods for improving performance. Section 2 shows that the speed of intra-site communication can be improved by 50% using special purpose local communications methods. In Raid this would lead to a reduction of 25-50 ms in transaction processing time. Section 3 shows that multicast can reduce the cost of inter-site messages substantially. For instance, a single message round-trip from the Raid level takes about 14 ms. A Raid level multicast to ten sites would take 35 ms. On the other hand, a low-level multicast facility could send the message to the ten sites in 3.5 ms + 0.5 ms per site (Table 2), for a total time of under 10 ms. This change would save 50 ms per transaction.

We are currently modifying the Raid communications package to make use of additional low-level services such as special-purpose local communications and multicast. The extension to support special-purpose local communications requires a new version of the select() system call that is able to detect
events that occur on multiple input facilities, some of which may not be devices. The extension to support multicast requires the notion of a multicast group in the communications package. Addresses must be assigned to these groups, and each process that is a member of a group must inform its kernel via an ioctl() that it is to receive messages sent to the group address. Since Raid supports multiple virtual sites running on a single physical site we will also extend the multicast input driver so that it can deliver a message to multiple processes on a single machine.

We are also investigating extensions to the Push system. Push is a general feature that can be extended to support functions other than network services. For instance, [1] uses user-level programs to extend the functionality of standard file systems calls. Blocks from a file are read by the kernel, passed to a user process, manipulated by the process, returned to the kernel, and finally passed to the destination user process. Since the implementation is entirely outside of the kernel, performance is a problem. The same functionality implemented in Push would only cross the kernel boundary once, yet users would still be able to customize the services offered by the file system to their own needs. Finally, we are investigating using a Push implementation of two-phase commit in RAID. Collecting responses in the kernel will further increase the performance advantage achieved with low-level multicast.

References