Distributed Process Groups in the V Kernel

DAVID R. CHERITON
Stanford University
and
WILLY ZWAENEPOEL
Rice University

The V kernel supports an abstraction of processes, with operations for interprocess communication, process management, and memory management. This abstraction is used as a software base for constructing distributed systems. As a distributed kernel, the V kernel makes intermachine boundaries largely transparent.

In this environment of many cooperating processes on different machines, there are many logical *groups* of processes. Examples include the group of file servers, a group of processes executing a particular job, and a group of processes executing a distributed parallel computation.

In this paper we describe the extension of the V kernel to support process groups. Operations on groups include group interprocess communication, which provides an application-level abstraction of network multicast. Aspects of the implementation and performance, and initial experience with applications are discussed.

Categories and Subject Descriptors: C.2.4 [**Computer–Communication Networks**]: Distributed Systems; D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Algorithm, Design, Measurement, Performance

Additional Key Words and Phrases: Distributed system, interprocess communication, job control, kernel, parallel computation, process group, servers

1. INTRODUCTION

The V kernel [11] provides an abstraction of processes, with operations for interprocess communication (IPC), process management, and memory management. This abstraction is used as a software base for constructing distributed

This work was sponsored in part by the Defense Advanced Research Projects Agency under contract N00039-83-K-0431 and National Science Foundation Grant DCR-83-52048.

This paper was nominated for publication in *TOCS* by the Program Committee for the ACM SIGCOMM '84 Symposium: Communications Architectures and Protocols. The committee was headed by Professor Gregor Bochmann. I would like to thank SIGCOMM and the referees who reviewed the paper in a timely way so that the full paper could appear in *TOCS* with an abstract in the symposium proceedings.—Editor

Authors' addresses: David R. Cheriton, Computer Science Department, EECS Department, Stanford University, Stanford, CA 94305; Willy Zwaenepoel, Computer Science Department, Rice University, Houston TX 77001.

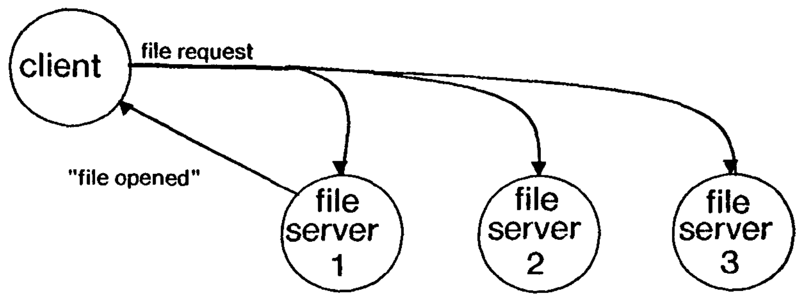

Fig. 1. Request to a group of file servers.

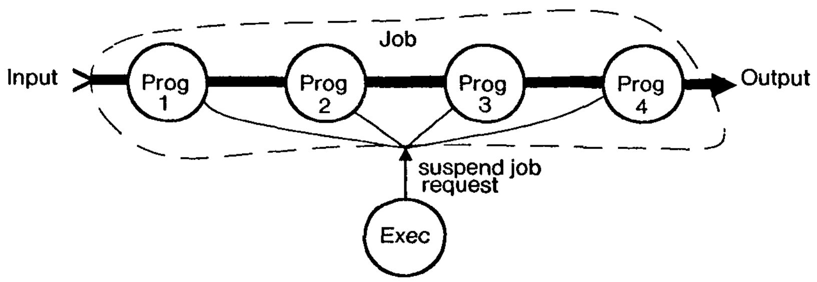

Fig. 2. Suspending a group of processes.

systems. As a distributed kernel, the V kernel makes intermachine boundaries largely transparent. In this environment of many cooperating processes, it is useful to be able to view some set of processes as a single logical entity, a process *group*, and perform operations on this entity.

As one example, a distributed system may have multiple servers providing file service. To locate a given file, a client process may need to communicate with the group of file servers to determine which one has the file, as illustrated in Figure 1.

Another example arises in executing a *pipeline* of filter programs. The user may wish to suspend, resume, or terminate the group of processes executing this pipeline, as depicted in Figure 2. This is an example of *job control*, controlling a set of programs in execution. Finally, in a distributed computation, process groups and group operations can be used for *intraprogram* communication and control. That is, when one logical program is executed as multiple processes (possibly executing across several processors), the set of processes executing the program can be viewed as a group. Group operations can then be used for intraprogram communication and control, as suggested in Figure 3.

In general, group operations cannot be implemented satisfactorily using multiple single-process operations for three major reasons. First, the identity of all the processes in a group may not be known to the invoker of the operation(s). For example, in a query to locate a file, the client may not know the addresses of

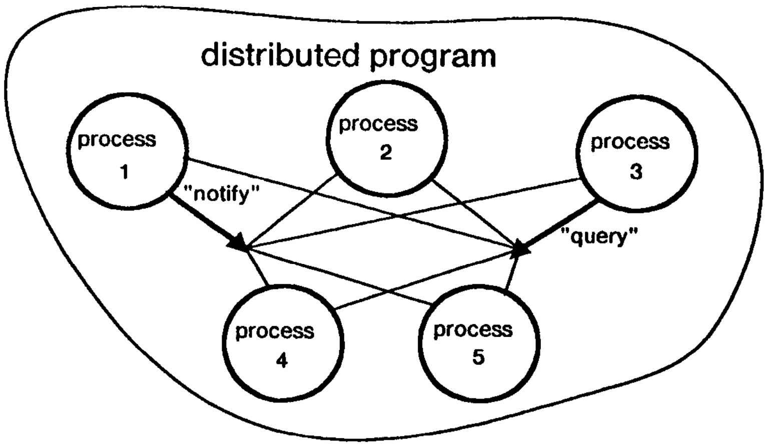

Fig. 3. Group communication in a distributed program.

all the file servers. With a *well-known* group identifier for the file server group and operations to communicate with a group, the client's query is straightforward. Second, using multiple single-process operations is less efficient than using group operations. For example, in Section 5 we describe a multiplayer distributed game in which group IPC operations are used for the real-time update of the replicated game state. In this case, group IPC operations are particularly efficient because the implementation makes use of the hardware broadcast and multicast facilities available on many local network and bus technologies, such as, for instance, the Ethernet [20]. This significantly reduces the number of network packets when the process group is large. Group interprocess communication operations provide an application-level abstraction of network multicast, thereby allowing applications to capitalize on the broadcast nature of the underlying network without developing their own network-specific facilities. Finally, multiple single-process operations do not provide the concurrency that is possible and desirable with group operations. For example, in a parallel distributed computation, one would like to transmit information to all the members of the group at the same time rather than serially.

In this paper we describe the extention of the V kernel to support process groups and group operations. We argue that this design provides a powerful facility for applications yet allows a relatively simple and efficient extension to the kernel. In Section 2 we describe V process groups and group operations. In Section 3 some aspects of the implementation, including some measurements of its space requirements in the kernel are discussed. Some performance measurements are presented in Section 4, and some initial applications of process groups are described in Section 5. In Section 6 we present the issues associated with process groups and group communication, and the rationale for the V kernel process group support. In Section 7 we relate our design to other work on multicast and group communication. We close with conclusions and problems for future study.

2. V PROCESS GROUPS

A V process *group* is a set of one or more processes, possibly on different machines. All processes in a group are equal; there are no distinguished members. A process group is identified by a group identifier, or *group-id.* Group identifiers are identical in syntax and similar in semantics to process identifiers. In V, each process is identified by a 32-bit *process identifier* or *pid* that is unique within a V domain. A *V domain* is a set of machines running the V kernel that implement a single system abstraction (typically spanning a collection of machines connected by a local area network). A group-id is a 32-bit identifier that uniquely identifies a group within a V domain.

A number of new kernel operations have been introduced to manage process groups. Additionally, most of the V kernel operations that take a process identifier as a parameter have been extended to perform the corresponding group operation, when given a group identifier as a parameter.

2.1 Group Management Operations

Processes can create, join, and leave groups using the following operations.

A new group is dynamically created by

```
group-id = CreateGroup(initialPid, type)
```

which returns a unique group identifier, with the process designated by `initialPid` becoming the first member of the new group. The `type` parameter specifies a group as *global* or *local,* and as *restricted* or *unrestricted.* A *global* group can have members on any host in a V domain, whereas the members of a *local* group must reside on the same host as the initial member process. A *restricted* group restricts membership to processes with the same user number (or authorization). A restricted group represents one *principal,* in the security sense of the word. An *unrestricted* group allows any process to join.

A group-id is considered deallocated and the corresponding group is considered nonexistent when the last member leaves the group. There is also a range of (statically allocated) reserved group-ids, disjoint from those allocated by `Creategroup`. The use of these reserved group-ids is similar to the *well-known* sockets in PUP [8] and other protocol families. The operation

```
JoinGroup(group-id, pid)
```

makes the process with process id *pid* a member of the group with group identifier *group-id.* A process may belong to more than one group. The operation

```
LeaveGroup(group-id, pid)
```

removes the process with process id *pid* from the group with group identifier group-id. The operation

```
QueryGroup(group-id, pid)
```

returns OK if the process specified by *pid* would be allowed to join the group specified by *group-id.* The return value also allows the caller to distinguish whether the process is already a member of this group, the group does not exist, or the process would not be allowed to join. `QueryGroup` also returns the number of processes in the group, which is only exact if no packets are lost (and no

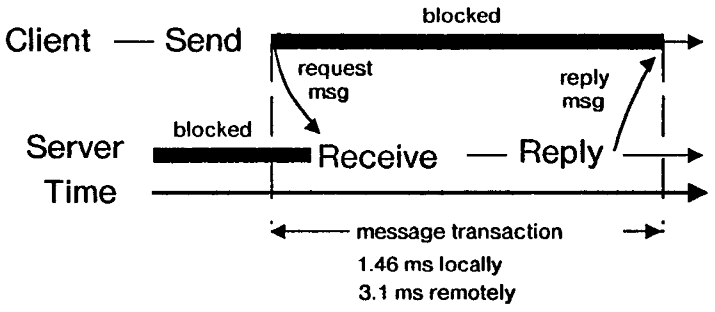

Fig. 4. Send–receive–rely message transaction.

processes join or leave the group during the execution of `QueryGroup`). As described later in the paper, determining the number of processes in a group, even just approximately, is useful in many applications.

Our goal of allowing a group-id anywhere a pid can be used can be applied to the group management operations as well. For instance,

```
JoinGroup(group-id1, group-id2)
```

adds all processes in `group-id2` in the group specified by `group-id1`.

2.2 Group Interprocess Communication

The basic communication model provided by the V kernel is that of processes communicating by *message transactions*. A message transaction is initiated by a *client* process executing `Send`, which transmits a *request* message to a *server* and blocks the execution of the *client* until a *reply* message is returned. Assuming that the *server* process is blocked waiting for a request message, the request message completes execution of a `Receive` operation, which reads the request message. The message transaction is completed by the server executing a `Reply`, causing a reply message to be sent back to the client. This sequence of events is illustrated in Figure 4, with the thick lines indicating blocked or suspended execution. As indicated in Figure 4, the time for a 32-byte message transaction is 1.46 milliseconds locally and 3.1 milliseconds between two SUN workstations connected by a 10-megabyte Ethernet.[1] The V system's IPC has been modeled after that of Thoth [10, 17]. The interested reader is referred to references [5], [11], and [15] for further discussion.

Sending to a group is similar to sending to a process, except that a group-id is specified instead of a process id. That is,

```
pid = Send(message, group-id)
```

[1] Previously reported performance figures [15] were 0.77 milliseconds locally and 2.56 milliseconds between two SUN workstations. The poorer performance figures in this paper are due primarily to extra checking and debugging facilities added to the kernel as part of a major reorganization. We expect the kernel performance to match the figures reported earlier once the reorganization is completed [12].

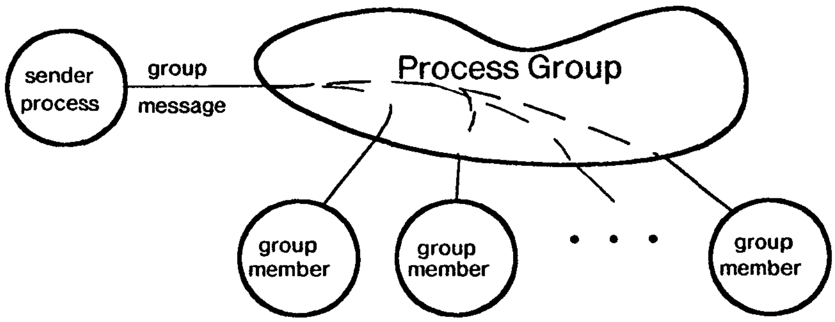

Fig. 5. Group message forwarded to group members.

sends a message to the group with group identifier `group-id`, which in turn forwards the message to each member in the group, as shown in Figure 5. The sender blocks until at least one process has received the message and sent back a reply using `Reply` (or until the kernel times out the message when there are no replies). The pid of the first process to reply is returned by `Send`. The first reply message overwrites the original message. Subsequent reply messages are received by the sender calling

```
pid = GetReply( replyMessage, timeout )
```

which returns the next reply message from a group `Send` in `replyMessage`, and the identity of the replying process in `pid`. If no reply messages are available within the time-out period, `GetReply` returns with `pid` set to 0. Additional reply messages for this transaction, if any, may be returned by further invocations of `GetReply`. It is thus left to the sender to decide how many reply messages are of interest and how long it is willing to wait for them. However, all replies for a message transaction are discarded when the process sends again, thereby initiating a new message transaction.

Both members and nonmembers of a particular group can send to that group. If the sender is a member of the group, the sending process does not receive a copy of its own message.[2] Any messages from a group that are queued for a process but not yet read are discarded if the process leaves the group by calling `LeaveGroup`.

`Receive` returns the next message, whether a group message or addressed to this specific process, returning the process identifier of the sender in both cases. By using the returned process identifier with `Reply` and `Forward`[3] the receiver can forward or reply to a message without knowing whether the message is the part of a group message transaction or not. However, the receiver can determine whether the message was sent to a group and, if so, to which group by checking the *forwarder* identifier associated with the message. If a group message, the

[2] This restriction avoids the deadlock problem of the sender blocking waiting for a reply from itself.
[3] `Forward` passes the request message to another receiver as though it was sent by the original sender to that other receiver.

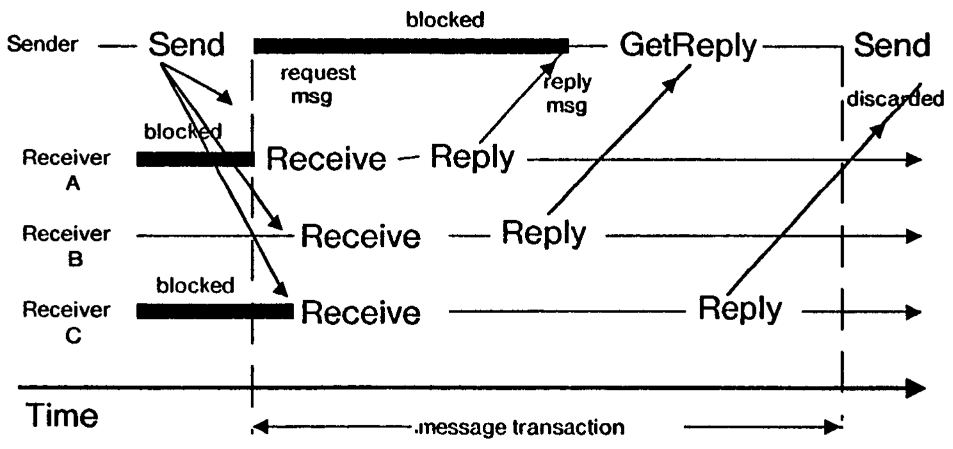

Fig. 6. Example of a group message transaction.

forwarder identifier is the group-id to which the message was sent. The operation

```
pid = ReceiveSpecific(msg, group-id)
```

suspends the invoking process until a message is received that was sent to the specified process group. The process identifier of the sender is returned in `pid`. `ReceiveSpecific` returns 0 if the process group does not exist and can in fact be used to wait for a particular group of processes to terminate.

For completeness, one can also specify a group-id to `Reply` and `Forward`. In this case, the operation applies to all messages that were sent to the receiver by sending to the specified group-id. In particular,

```
Reply(msg, group-id)
```

replies to every message that is awaiting reply from the invoker, having been sent originally to the specified group-id.

We have also been experimenting with a real-time `Send`. The sender does not block because no reply is allowed. Also, the message is sent as an unreliable (best efforts) datagram. In fact, the message is only received if the receiver(s) are ready to receive the message immediately when it arrives. Real-time send operations also work uniformly for both sending to a single process as well as to a group.

A typical group message transaction is illustrated in Figure 6. In this case, the sender sends a message to a group of three processes, A, B, and C. Two of the receivers, A and B, reply to the message in a timely fashion. The reply from A unblocks the sender, completing the execution of `Send`. The reply from B is received using `GetReply`. The third reply message (from C) is discarded by the kernel once the sender initiates a new message transaction, by calling `Send` for the second time. Note that a message transaction in this case is terminated by a new `Send`, not the reply as previously for one-to-one messages. Thus it is less meaningful to talk about the elapsed time for a message transaction. However, the time for the sender to receive the *first* reply is the same as the times cited for a one-to-one message transaction. It is the delivery of the additional replies that introduces further overhead.

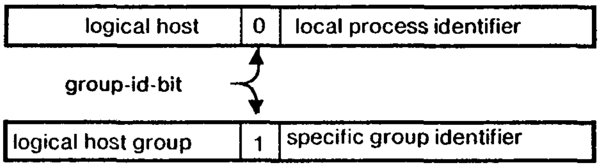

Fig. 7. Process identifiers and group identifiers.

2.3 Other Group Operations

Other V kernel operations that take a process id as a parameter have been extended to take a group-id as well. For example,

```
DestroyProcess(group-id)
```

destroys all processes in the group specified by group-id (subject to the usual permission requirements). Similarly,

```
ForceException(group-id)
```

causes all processes in the group to be suspended under the control of the V exception-handling mechanism (and possibly resumed later).

In summary, support for process groups has required adding the four group management operations plus `GetReply` to the kernel interface as well as extending the semantics of a number of the existing kernel operations to their group form. The principle of allowing a group identifier as a parameter in place of a process identifier wherever the latter occurs leads to some extended semantics that are sensible yet of limited utility. For example, one can imagine using the group `Reply` operation in a distributed computation to reply to a set of helper processes, providing them all with the same information at the same time. However, we do not expect group `Reply` operations to be frequently used.

3. IMPLEMENTATION

The distributed V kernel executes as multiple instantiations of the same kernel code, one per participating machine. The different instantiations communicate through a low-overhead interkernel protocol to provide a single-kernel image [15]. The V kernel has been implemented on SUN workstations [3] connected by a 10-megabyte Ethernet [20] or a 3-megabyte Ethernet [26]. In this section we describe the interesting aspects of the process group implementation in the V kernel.[4]

3.1 Group Identifiers and Group Membership Records

A group-id is a 32-bit identifier, similar to a process identifier except that the *group-id bit* is set to 1 rather than to 0, as illustrated in Figure 7. The group-id bit allows the kernel (and discerning processes) to efficiently distinguish between group identifiers and process identifiers. It also ensures that the group and

[4] The implementation described here is independent of the SUN workstation architecture, and except as noted, independent of the use of the Ethernet.

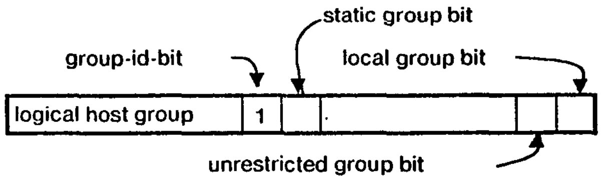

Fig. 8. Group identifier subfields.

process identifier name spaces are disjoint. Bits within the lower order 16-bit word of the group identifier are used to identify whether the group is a *static* group, that is, it has a statically allocated group identifier, and whether the group is *local* or *global*, and *restricted* or *unrestricted*, as shown in Figure 8.

A group identifier encodes in its high-order bits the *logical host group* for this group, the set of hosts on which members of this process group reside. The low-order bits indicate the specific group within this logical host group. Similarly, a process identifier encodes in its high-order bits its *logical host*, identifying the host on which it executes. When the group-id has the local group bit on, the logical host group subfield is interpreted as a logical host identifier, the same subfield as in a process identifier.

Each instantiation of the kernel maintains a hash table accessed by group identifier recording the group membership of all processes local to this kernel. For each group membership of a local process pid, there is a record containing the (group-id, pid) pair. The membership information for a group is distributed across all the hosts that have members in the group. In particular, each kernel only maintains information about its own processes. This requires less space, network traffic, and code complexity than replicating the group membership information in all hosts and prevents inconsistencies between different records of group membership. However, it does rely on an efficient mapping of a group-id to logical host group and from a logical host group identifier to hosts within this host group. Mapping a group-id to a logical host group is trivial because the host group identifier is encoded in the group-id. Mapping a logical host group identifier to host within this group is network dependent but uses the following general structure.

The V kernel uses a transport-level, network-independent packet format specified by the V interkernel protocol [15]. On transmission, an interkernel packet is embedded in a network-level or internetwork-level datagram. The datagram is assumed to specify host-level addressing as well as provide error detection on the packet during transmission. At the internal kernel interface to network-specific code, the kernel passes an interkernel packet to the network driver. The network driver translates the process-level addressing in the interkernel packet into host level addresses, which it places in the network-level header, as illustrated in Figure 9. Rather than insist that the kernel know the addresses of all logical hosts, the kernel only maintains a cache of recently used logical hosts. If a logical host mapping is not known, the kernel uses the host group address corresponding to all V kernel hosts. This corresponds to broadcasting to all V kernel hosts within the V domain of machines.

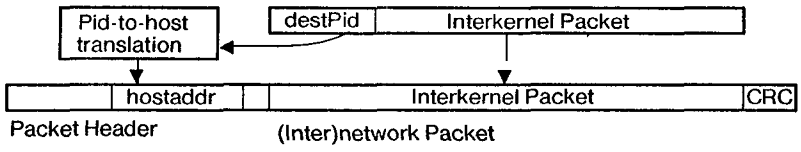

Fig. 9. Interkernel packet in a network packet.

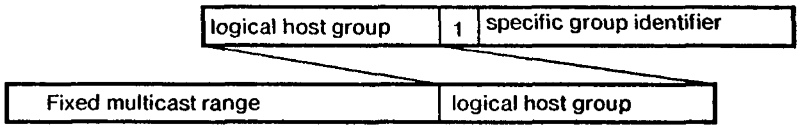

Fig. 10. Group-id to Ethernet multicast address mapping.

For group messages, the kernel extracts the logical host group from the group-id associated with the message and addresses the packet to the datagram address corresponding to this host group. For the 10-megabyte Ethernet, the logical host group is mapped to a multicast address by prepending a fixed high-order address portion to the logical host group identifier, as illustrated in Figure 10. This procedure maps group identifiers to one of 2^{15} preassigned Ethernet multicast addresses used by the V kernel. In this way, the kernel can determine the multicast address corresponding to a group efficiently from the group identifier. Moreover, the network interface can reject all packets addressed to logical host groups of which the kernel is not a member.

Group operations are implemented in the two ways. The group IPC operations are implemented directly as low-level extensions of the corresponding single-process IPC operations. The other operations are implemented using the group IPC operations.

3.2 Group IPC Operations

The primary extension for groups is to the message sending mechanism. A `Send` to a group identifier is recognized in the kernel by testing for the group-id bit. If the group-id bit is 1, the kernel checks for local group members and delivers the message to each one of them. It then transmits a message packet addressed to the logical host group corresponding to this group. The kernel executing on each of the hosts in this host group is then responsible for delivering a copy to each of its local member processes.

The packet is retransmitted for some maximum number of times until the first reply message is received. *Reply-pending* packets are returned when a group member has received the message but not replied yet, as for single-process message transactions. The first reply unblocks the sending process; subsequent replies are queued, provided that they arrive before the beginning of the next message transaction. The reply queue is used by `GetReply` to return additional reply messages, and emptied by `Send` at the start of the next message transaction. The current message transaction is indicated by a *message transaction number* stored in the process descriptor.

When the kernel receives a packet addressed to a group, it delivers the message to each local member in the group, as indicated by the local group membership table. The copies of the group message are buffered in the same buffer records that are used to buffer remote one-to-one messages.

The remote message buffers are discarded after a time interval if the sender starts a new message transaction before the message has been replied to. This prevents a receiver from processing a message associated with an *arbitrarily* old message transaction, although it does not prevent a receiver from processing a message associated with a terminated message transaction. It is simply too expensive to garbage collect all outstanding copies of a message from a previous group `Send` when a new message transaction is started.

`ReceiveSpecific` is implemented by checking for messages that were sent to the specified group-id, as in the single-process case. It also periodically transmits a query packet to check that there are members in the group and expects to receive confirmation, timing out otherwise. In this way, the mechanism for `ReceiveSpecific` on a group is a simple extention of that for a single process. Note that `ReceiveSpecific(msg, group-id)` only receives a message sent to the specified group-id, not a message sent by a process in this group to the receiving process. The latter semantics are less useful and more difficult to implement because they require local knowledge of remote group membership.

`Reply` and `Forward` with group-id are similarly simple to implement: They simply reply to or forward each process that is awaiting reply from the receiver as a result of sending to the specified group id. In this vein, they are simply repeated applications for the single-process operations and offer little performance benefit.

3.3 Other Group Operations

The kernel process management, group management, and memory management operations are implemented by a server process internal to the kernel, called the *kernel server*, allowing these operations to be invoked using the standard IPC facilities. For example, `LeaveGroup`, as invoked by the application, is actually a small *stub routine* that sends a message to the local kernel server requesting this operation. There is a kernel server process in each instantiation of the kernel in a V domain. All kernels servers belong to the *kernel server group.*

To implement a group operation that requires actions on multiple hosts, the request is sent to the kernel server group, rather than to an individual kernel server. For example, `DestroyProcess(group-id)` results in a message being sent to the kernel server group. Each kernel server destroys the local processes in the specified group, assuming that the requestor has the required permission. It does this by invoking the single-process destroy routine on each local process in the group. Similarly, `QueryGroup` sends to the kernel server group to get all the information on the specified group.

The semantics of these group operations are complicated by the fact that a group operation can be partially successful. For instance, `DestroyProcess` on a group may destroy some but not all processes because the requestor has no permission on a few of the processes in the group. Generally, these operations indicate failure if they detect a failure; that is, success is having no failures, not just some successes.

Each one of these stub routines must handle the unreliable nature of the group interprocess communication. For example, it is possible that not all kernel servers in the kernel server group receive a destroy process request, and so not all processes in the group are destroyed as a result of a single group send. To handle this, the `DestroyProcess` routine uses `QueryGroup` to determine whether there remain any group members after sending the destroy request. If so, it reissues the destroy request. Of course, if a reply from a kernel server indicates no permission or any other reason for failure, `DestroyProcess` returns the reason for failure.

The kernel servers use a standard procedure in responding to group requests. If a group request refers to a process, group, or other object that it has no knowledge of, it discards its reply; othewise, it replies as usual. For example, in response to a `QueryGroup` request, only kernel servers with local members in that group respond. This reduces the number of reply messages that a client must deal with to those with useful information. It also means that querying a nonexistent group results in a significant delay (just under 1 second) before the kernel times out the message and indicates that no responses were received. Thus with `DestroyProcess` as described above, the time to check that all processes in the group have been destroyed considerably increases the time for this operation. In general, the most efficient means to the (reliability) semantics for each operation seems specific to the operation and to the group semantics we deem most useful for the operation. The issue of efficiency and reliability with group operations is discussed further later in the paper.

`JoinGroup` is implemented by sending a message to the kernel server group. If none of the kernel servers object to the request or at least one returns approval, it is considered successful and the join takes place. `QueryGroup` is implemented by collecting replies from all the kernel servers. Each reply specifies the number of processes local to the replying kernel server that belong to the specified group. `CreateGroup` is implemented purely as a standard library routine, using `JoinGroup` to join a randomly generated group-id as the group's first member and `QueryGroup` to determine that it is in fact the first member. Otherwise, `LeaveGroup` is executed to remove the process and a new group identifier is regenerated.

Joining groups and creating new groups require *distributed agreement* [23] among the instantiations of the kernel to decide whether joining is allowed or whether a group-id is unique. Our current implementation is not complete in the sense that it does not handle faulty behavior among the cooperating hosts. In particular, if a kernel server fails to reply to the join or create request, or if the message or its reply are not received, the operation is performed incorrectly. However, the use of group IPC to check with all the kernel server processes provides a simple implementation for achieving agreement that works well in practice. In particular, the group `Send` packet is retransmitted several times until either a kernel server responds positively or negatively or it is fairly certain that the process is allowed to join the group (or the group is new). A consequence of using several retransmissions is that joining an existing group typically takes slightly over 3 milliseconds, while joining a nonexistent group or creating a new group takes 0.75 second, our current time-out period including retransmissions.

3.4 Portability

We endeavor to make both the design and the implementation of the V interprocess communication reasonably portable to other networks besides the 10-megabyte Ethernet. The concern for network portability includes the group IPC extension. V Group interprocess communication is implemented on the basis of a facility to communicate with groups of hosts or *host groups.* Logically, a process group defines a *host group* as the set of hosts executing processes in the given process group.

The host group abstraction divides the portability discussion into the consideration of two issues: the requirements that host groups must satisfy to serve as a base for the implementation of V group IPC, and the problem of implementing host group support that satisfies these requirements on different networks and internetworks.

3.4.1 *Host Group Requirements.* The V kernel group IPC requires the host group facilities to satisfy the following requirements:

(1) *Domain Host Group.* For each V domain, there must be a host group to which all V kernel hosts in that domain belong.
(2) *Addressing.* A single packet address should suffice to address a packet to a host group. A host group address should be similar to a single host address. Ideally, it should be possible to encode the host group address in the group identifier.
(3) *Reliability.* Transmission to members of a host group need not be totally reliable. However, a small number of retransmissions must ensure (with very high probability) that each host in the group receives at least one copy of the transmission. There is no logical problem (only performance loss) with a machine receiving multiple copies of a packet or receiving packets not addressed to it.

We also assume there are statically allocated (well-known) host group addresses, such as the domain host group, or some means of allocating and registering group addresses to simulate well-known host group addresses.

The minimal requirement is satisfied by having one domain host group that meets these addressing and reliability constraints. The ideal is the provision of many host groups with both statically and dynamically allocated host group addresses.

For a secure communication network, either encryption should be available or it must be possible for the network to prevent an unauthorized host from joining a host group. Of course, the host must not be able to receive packets not addressed to it, including packets addressed to groups of which it is not a member. Alternatively, the network and all copies of V kernel could be assumed to be secure, in which case protection is imposed collectively by the V kernel.

A host group implementation failing to meet these basic requirements would require modifications for the V kernel above the network module level. While these requirements are specific to the V kernel, we argue that they are reasonable requirements for other systems of similar functionality. The requirement of a domain host group ensures that all cooperating hosts can be reached as a group so that agreement can be obtained on allocation of new group identifiers, membership in groups, and the like. It also provides for the "worst-case" process group, in which there is a group member on every cooperating host. The addressing requirement eliminates the need for a sending host to know all the members of a group in order to send to the group. Otherwise, each host would have to maintain a list of member hosts for each group, and in particular, a list of all V hosts for the domain group. This would impose additional code complexity and execution cost in the kernel. The reliability requirement simply precludes one or more host members from systematically failing to receive group messages. Without this, retransmission is not guaranteed to ensure delivery even when all receiving hosts are functioning.

3.4.2 *Implementation of Host Groups on Different Networks.* There are two major aspects to the implementation of host groups:

(1) allocation of host group identifiers;
(2) delivery of packets addressed to host groups to all hosts in the group.

The addressing requirement for host groups implies that some portion of the space of host addresses must be reserved for host group addresses. We briefly consider these issues in broadcast networks and store-and-forward networks and internetworks below.

In a broadcast network, each host receives every packet at the data-link level and the network interface, and higher level software discards packets not wanted by the host using a series of so-called *packet filters.* Thus to implement host groups on a broadcast network requires a range of host addresses that can be used as host group addresses and modification to the packet-filtering mechanism to allow a host to receive packets addressed to specified host groups as well as to its own host address. This is basically the 10-megabyte Ethernet design [20]. The experimental 3-megabyte design is a degenerate case with a single-host group address corresponding to all hosts, thus providing the domain host group. A single domain host group (or broadcast address) is feasible in low-to-moderate speed local networks with infrequent use of group communication. Additional filtering is then implemented in software. However, at high speeds the processing cost of receiving and discarding unwanted packets becomes considerable.

For store-and-forward networks, and by extension for datagram internetworks, host group addressing can be similar. That is, a subrange of host addresses is reserved, or in an internetwork, a subrange of network numbers is reserved. However, the delivery mechanism of the network needs to be extended to be able to deliver a single packet to multiple hosts. Several techniques for efficient delivery of broadcast and multicast packets in a store-and-forward network have been studied by others [19, 32] and provide realistic solutions to host group delivery. The major practical problem with implementing this scheme in an existing network seems to be the limited space for extra code and tables in a typical switching node or gateway.

A subtle portability problem arises with the allocation of group identifiers because they each include a logical host group identifier. Currently, a dynamically allocated group identifier is selected at random, resulting in a random selection of logical host group identifier. On a different network, with a higher cost per logical host group (such as a point-to-point network), there may be greater benefit choosing logical host groups more carefully. There is a trade-off between the number of host groups and redundant messages. At one extreme, there is one process group per logical host group, and network hosts only receive packets sent specifically to them or to a logical host group to which they belong. Thus no processor time is lost discarding packets not intended for the host,[5] but more space may be required to record all the logical host groups. At the other extreme, all process groups use the same logical host group, minimizing the number of host groups but causing every host to receive all messages sent to a group, independent of whether it has processes in that group or not. This is basically the implementation strategy that we use on our 3-megabyte Ethernet. On a store-and-forward network, this would result in many redundant packets.

3.5 Implementation Summary

We have implemented the group management, group interprocess communication and a few of the other operations (including `DestroyProcess`). This code has added about 3 kilobytes of code to the kernel as well as requiring an additional 12 bytes in every process descriptor for the reply queue. (The total size of the kernel is now 52 kilobytes when configured to support a maximum of 64 processes.) We have yet to remove all the mechanism rendered redundant by the addition of the group support. For example, the mechanism that allocates a logical host number during kernel initialization originally used a special broadcast mechanism to check for a collision with an existing logical host number. This procedure is being changed to use a group `Send` to the kernel server group, thereby eliminating some special-purpose code in favor of the more general group IPC mechanism. Similarly, a simple name-mapping mechanism provided by the kernel now uses group IPC rather than its old special-purpose broadcast mechanism.

The addition of group IPC imposes a time cost on the speed of one-to-one interprocess communication in flushing the reply queue and checking for the group-id bit during the execution of `Send`. This overhead adds about 17 microseconds to the local message transaction time, an increase of 2.2 percent.

Overall, the implementation is fairly compact, has had minimal impact on the performance of the more common one-to-one IPC, and has eliminated the need for some rather awkward mechanisms within the kernel, as mentioned above.

4. PERFORMANCE MEASUREMENTS

The group IPC operations are the most performance critical of the group operations both because they occur directly in the most performance-sensitive uses and because they are used to implement the other group operations. We have done some preliminary performance measurements of the group IPC. The elapsed time for a group `Send` is dependent on a number of factors, including the number of members in the group, the number of members that reply to a message, the time the sender is willing to wait for replies, and whether some of the

[5] Note that few Ethernet interfaces allow the processor to select exactly the multicast addresses that it wishes to receive.

Table I. Elapsed Time (milliseconds) with Packet Loss

	Group Transactions with Packet Loss: Number of replies			
Members	1	2	3	4
1	3.36	—	—	—
2	4.46	5.86 (3.8)	—	—
3	4.61	5.47 (1.9)	18.50 (30.0)	—
4	4.78	5.12 (0.1)	14.32 (18.0)	24.34 (45.0)

members are local or not. The elapsed time is also dependent on the speed of the processor and speed of the network interface. For our measurements, we use 10-megahertz 68010-based SUN workstations connected to a 10-megabyte Ethernet by an Ethernet interface with two receive buffers. This configuration is similar to the one used for our earlier measurements of one-to-one IPC performance [11]. When used as the sending machine in a group message transaction, this machine configuration suffered from severe packet loss for three or more members in a group. Packet loss occurs because both receiver buffers in the Ethernet interface are full when another packet is sent on the network, so that the interface ignores this packet. We also report measurements on an 8-megahertz 68000-based SUN workstation with a 10-megabyte Ethernet interface with heavy buffering. This configuration does not suffer from packet loss (at least in our measurements). Unfortunately, this Ethernet interface is significantly slower in host-to-interface operations, thus (in combination with a slower processor) giving much poorer performance.

The measurements were made by performing N times a group Send and receiving a number of replies, and dividing the elapsed time by N to get a reasonably accurate elapsed time for a single operation. Table I gives the time for a group message transaction as a function of number of remote group members and number of replies received for the first machine configuration. Note that all remote group members reply; the number of replies indicated in the table is the number of replies that the sender attempts to receive. The figures in parentheses indicate the percentages of reply messages that are lost of those that we try to receive. For example, with four group members and receiving three replies, out of 10,000 group Send operations, only 24,398 replies are received, whereas we should have received 30,000 with no packet loss.[6] A lost reply packet inflates the elapsed time by the time lost waiting for a reply packet to appear, 20 milliseconds in these measurements.

The first observation is that packet loss becomes significant with three or more group members. For example, with three group members, only 70 percent of the expected replies were received. This behavior is understandable given that each group member in this experiment is replying immediately. Thus reply packets are being returned to the sending machine as a series of back-to-back

[6] This does not indicate the number of packets actually lost because, for example, if one reply packet was lost with four group members and the sender waiting for three replies, the lost packet would not be noticed.

Table II. Elapsed Time (milliseconds) without Packet Loss

	Group Transactions without Packet Loss: Number of replies				
Members	1	2	3	4	5
1	8.87	—	—	—	—
2	12.23	13.10	—	—	—
3	16.26	16.27	17.27	—	—
4	20.25	20.30	20.31	21.57	—
5	24.28	24.32	24.42	24.45	25.92

packets. With only two receive buffers, the sending machine cannot process the incoming packets fast enough to avoid dropping packets. (The packet loss with only two group members is explained by other random broadcast and multicast traffic on the network during the experiments.) Thus the figures for three- and four-group members are irrelevant in indicating elapsed time. Until we have Ethernet interfaces with more receive buffers, we have modified the kernel to introduce a random delay when replying to a group `Send`, thereby reducing the packet loss problem but increasing the elapsed time.

Comparing a group send with one reply to a one-to-one message transaction, with one group member, the elapsed time is approximately 260 microseconds greater for a group `Send`. This is accounted for by the time to check for local group members and a slightly more complicated delivery mechanism in the receiving kernel. Otherwise, the mechanism is the same.

The cost of a group `Send` increases with more members in the group. Considering the difference between one and two group members, it appears to cost 1.1 milliseconds to receive a reply packet, queue the reply, and discard the reply on the next message transaction. The cost for each additional group member appears to be lower for three- and four-group members. However, the measured cost in these cases is reduced due to loss of reply packets. It may also be reduced by reply packets arriving after the next message transaction has started, in which case the old replies are discarded immediately.

To explore the performance without packet loss, we use an interface with a much larger Ethernet receive buffer pool. These measurements are given in Table II. Because the machines used for the measurements in Table I and Table II run at very different speeds, we do not draw any comparisons between the two tables.

In Table II, it is clear that without packet loss, each additional group member imposes an extra cost for processing its reply packet, whether or not it is read by the sending process (using `GetReply`). In fact, the cost of reading all K replies is only slightly higher than reading only one and discarding the remaining $K - 1$. We use this measurement later to support the provision for reading multiple replies (see Section 6). One should note that the elapsed times for group message transactions, as presented in Tables I and II, include the cost of receiving all replies, regardless of whether they are read by the sender or not. In practice, if not all replies are read, the cost of discarding replies is incurred concurrently or subsequently at the interrupt level or as part of the next `Send` operation.

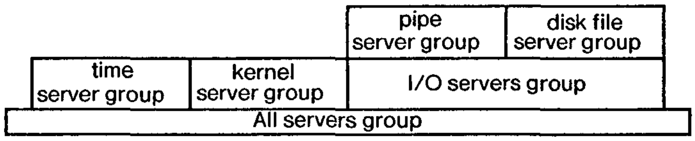

Fig. 11. Server group hierarchy.

The cost of other group operations is basically the cost of the group message transaction with the kernel servers, the cost of the kernel servers (in parallel) performing the operation, and the cost of ensuring the necessary (reliable) semantics on the operation. As mentioned earlier, the latter can easily dominate the cost.

In general, these measurements indicate that V group IPC is practical to use in applications in which a communication overhead of 10–30 milliseconds is acceptable (depending on the size of group and machine speed, etc.) Related to this, the number of Ethernet receive buffers is a critical factor in determining the number of reply packets that are lost and therefore in the cost of many of these operations.

5. APPLICATIONS

We identify three major uses of process groups, namely server groups, job control, and parallel computation. In this section, we describe some specific uses of process groups in each of these generic categories.

5.1 Server Groups

The V system servers have been organized into groups according to the service that they provide. Logically, there is a hierarchy of server groups in our system starting with the group of all server processes. Subgroups of this universal server group include the group of all servers supporting the I/O protocol [13], which in turn contains as a subgroup the group of all disk-based file servers. A subset of the V server hierarchy is depicted in Figure 11. The server groups include kernel servers, file servers, pipe servers, time servers, and team servers.[7] Group hierarchies are implemented by processes belonging to multiple groups. There is no explicit support provided or (seemingly) needed for group hierarchies.

As described earlier, the kernel server group allows the kernel to make use of the group mechanism to implement the non-IPC group operations.

The team server group is used to locate unloaded workstations as part of a distributed global scheduling mechanism. In this use, a request message is sent to the team server group specifying a lower (and upper) bound on memory and processor availability (plus possibly other requirements). Only team servers on machines satisfying these constraints respond.

We have just begun experimenting with server groups for *decentralized* name mapping. That is, rather than going through a central name-mapping facility,

The *team servers* manage the execution of programs, consisting in V of *teams* of processes.

the name and its associated request message are sent to the appropriate server group for the request, and those servers that recognize the name respond. This is one area where unrestricted groups are of use because, with the uniform service protocols used in V [13, 14], the group of servers that might be involved in handling a particular request does not necessarily correspond to a single user or a single principal in the security sense of the word.

Another potential server application arises with distributed two-phase commit of atomic transactions [24]. The transaction coordinator uses a group message to issue a "prepare-to-commit" command to the group of all transaction servers. Each transaction server then responds with a reply message indicating whether it is prepared to commit or not. Finally, the transaction coordinator issues a "commit" or an "abort" command, again using a group message to all transaction servers. We note that in the "prepare-to-commit" phase, the group message from the coordinator to the transaction servers must be delivered in a reliable fashion to all transaction servers, and a reply from each transaction server is necessary. In contrast, the final commit message does not need that degree of reliability, because transaction servers can be expected to check back with the transaction coordinator if they do not receive the final message in due time.[8]

5.2 Job Control

We are currently modifying our command interpreter to use process groups for job control. All processes executing a single job, for instance, a pipeline of filter programs, belong to the same process group. The command interpreter can suspend, resume, or terminate the job by invoking the corresponding operations on this group. It can also wait for the job to terminate by executing a `ReceiveSpecific` on the associated group-id. Note that there is no requirement that all processes in a job execute on the same host although the provision of local groups allows one to take advantage of this special case when it arises.

5.3 Distributed Parallel Programs

The group IPC facility is being used by distributed programs that run in parallel on several workstations.

Real-time group communication (see Section 2) is used in a version of Amaze [4], a multiplayer game program that runs on a set of networked personal workstations running the V kernel.[9] In Amaze, the database describing the state of the game is replicated across all the participating workstations. Updates to the state are sent periodically to the group of "game manager" processes, one per player, which incorporate the update into their respective copies of the game state database. Update messages are designed so that if a message is lost, a subsequent update message subsumes the data of the lost message. In fact, each

[8] This mechanism assumes that a transaction is suspended until the transaction server learns from the coordinator whether the transaction was aborted or finally committed. In particular, if the coordinator fails after the "prepare-to-commit" message and before the server receives the final commit message, the transaction commits or aborts only after the transaction coordinator recovers and makes the final decision known.

[9] The original version of Amaze used one-to-one IPC. The version using group IPC generates less load on the network but does not behave any differently from a user's perspective.

workstation extrapolates the game state forward according to its current copy of the state, with update messages indicating changes to the *first derivative* of the state plus correct information. The interested reader is referred to the Amaze report [4] for further details. The V kernel group IPC facility allows Amaze to be implemented independently of the underlying network technology or machine and executed in parallel with other applications running on a workstation. Thus, while Amaze is similar to games like Maze Wars on the Xerox Alto computers, it does not require direct and exclusive access to the Ethernet interface and it does not contain any network-specific code.

Another application of group IPC arises with a checkers playing program that runs multiple move evaluator assistants on multiple machines, implementing a parallel α–β search. In this program, group IPC is used to exchange search information between searcher processes, thereby reducing and focusing the search effort. Similar use can be made of group IPC in implementing, for example, the traveling salesman problem. Here the current best path is communicated to all parallel searchers, reducing the exploration of inferior routes. We are also developing a concurrently executing rule-based system in which group IPC is used to communicate the subgoals that have been resolved, again to avoid redundant computation.

In general, in distributed computations that do not use shared memory, group IPC appears to be a valuable mechanism for querying the progress of the other processes taking part in the computation, as well as for notifying other processes of new insights gained. Because we recognize the trade-off between redundant computation and communication overhead in distributed computation, we look to group interprocess communication as a promising mechanism for reducing the cost of "global communication" in distributed programs.

6. DESIGN ISSUES

In extending the V kernel to support process groups, we recognize the following:

—Group operations should be a natural extension of the single-process operations, wherever possible.
—Group operations should be efficient (especially on broadcast networks.)
—Group operations should support a wide range of applications with minimal additional complexity added to the kernel.

The first point was addressed by making group identification compatible with process identification and allowing a group identifier to be used (almost) anywhere that a process identifier can be used. This approach has the benefit of simplifying the code that uses the group facilities, simplifying the V kernel interface, and avoiding redundant code in the kernel itself. The second two points lead to careful consideration of issues in reliability, performance, and security, as described below.

6.1 Reliability

We define *reliable group communication* to mean that at least one member of the group receives and replies to the message (or else a failure indication is returned). This definition of reliable transport, as an extension of the transport-level delivery provided by the one-to-one IPC, results in a minimal cost implementation in the kernel. It is also the minimum facility required for successful query and reliable notification. However, some applications require more reliable communication than this and should be able to implement it using the provided group mechanism. We define a group facility to be *k-reliable* if either at least *k* members of the group are guaranteed to have had the operation performed on them or else a failure is signaled to the sender. We use *all-reliable* to designate guaranteed operation on all members of the group. In this terminology, the V kernel implements 1-reliable (and 0-reliable) operations directly and provides facilities for applications to implement *k*-reliable and all-reliable operations.

This design has several advantages over providing all-reliable group operations directly in the kernel. First, an application can implement the level of reliability of group operations that it requires and does not have to pay for reliability that it does not require. For example, all-reliable group communication is not needed by, and is too expensive for, many applications, including distributed games, service queries, and advisory notifications. Second, an application can use application-specific knowledge to implement the required degree of reliability more efficiently than the kernel can, without the benefit of that knowledge. For example, the query operation may need replies from some percentage of the servers in a group but not necessarily all. The client can reasonably know the percentage and the time that is reasonable to wait for these replies to return. Finally, our design allows a simpler kernel. For example, all-reliable group communication requires that either the receivers ensure that they receive each message using a stable message logging facility [9], or, more conventionally, the sender know the exact membership of the group and get positive acknowledgments from each member of the group. In the latter case, the use of statically allocated group-ids to locate services would be significantly restricted if the kernel must know the group membership in advance. Moreover, maintaining group membership information in the kernel and handling retransmissions efficiently would significantly complicate the kernel code and data structures.

Note that 0-reliable communication is provided because it is easy to implement and because it provides the required nonblocking semantics and efficiency for real-time applications. This is true for both the one-to-one and group real-time `Send` operations. Examples include distributed game programs, distributed monitoring, and real-time group communication in general.

Implementing all-reliable group operations in general entails providing all-reliable group communication. There are two basic approaches that an application can take to implementing reliable group communication, depending on whether one places the onus on the receiver or the sender for reliable delivery.

Putting the onus on the receiver for reliable delivery leads to what we call *publishing.*[10] It is so named because it mimics real world publishing. That is, information to be sent to a group, the *subscribers,* is filtered through the *publisher,* which collates and numbers the information before issuing it to the subscribers. A subscriber noticing a missing issue by a gap in the issue numbers or a new issue not being received in the expected time interval requests the *back issue*

[10] This is unrelated to the use of the term by Powel and Presotto [28].

from the publisher. Thus, instead of automatic retransmission until the receiver acknowledges the message, the receiver must request retransmission if it is required.

As a slight variant of this scheme, a message can be sent to a *logging process* that, by arrangement, is the only replying process, thus ensuring reliable delivery to the log. Then, subscribers request back issues from the log. This allows multiple contributors sending directly to the group but complicates the allocation of serial numbers.

The basic technique of publishing, requiring the receiver to ensure reliable delivery, makes it difficult to guarantee timely delivery of messages at a reasonable cost. A receiver can only discover that it has missed a message by receiving a subsequent message or by periodically checking with the publisher or logging process for missed messages. With infrequent published messages, either the receiver must incur the overhead of checking frequently with the publisher or logging process or else one must accept that lost messages may not be discovered for some time.

Publishing is useful for notifications but does not help with queries, where the main problem is getting all (or sufficiently many) replies back to the sender. Queries necessitate the second approach, in which the onus is put on the sender to implement reliability. Here, the sender resends the message to the group until it receives replies from all members in the group. When using our IPC primitives, this requires that the requested operation be *idempotent* because each resending of the group message appears to the receivers as a new message transaction. Thus several receivers may receive the message multiple times. We are considering providing a `Resend` operation to allow a client to retransmit as part of the same message transaction and use the existing duplicate detection and filtering mechanism in the kernel. While simple to implement, this introduces further timing dependencies at the application level because message transaction records are discarded after a certain time period of inactivity.

As an optimization, a sender can resend to the individual processes from which it failed to receive a reply, assuming that the sender knows the membership of the group. This reduces the load on other receivers in the group. It also avoids the problem that a group resend has of generating the same flurry of reply packets that caused the original reply packets to be dropped, possibly leading to a systematic error. Finally, it allows the client to exploit the positive acknowledgment and retransmission for reliable delivery (or determination of failure) of the one-to-one IPC.

The provision of multiple replies is an essential feature of our group operations for implementing all-reliable operations when the onus is on the sender for reliable delivery, as described above. The use of multiple replies is also important for operations with k-reliable communication. In particular, providing multiple replies allows an application to choose the number of responses that it requires to a query. For example, a query to locate available file servers requires multiple replies being returned, one from each file server. Even in just trying to locate a single file server, one reply may not be sufficient in the presence of a faulty file server that responds quickly to queries but fails on other operations.

Multiple replies can also be used as acknowledgements to notifications. For example, suppose that at least *k* processes in a group must receive a notification message to match the reliability constraints of a particular application.

The alternative design is to provide only one reply in a group message transaction. While this approach would simplify the kernel interface and the implementation of group IPC, it is overly restrictive and does not result in any significant performance improvement over allowing multiple replies. This is easily seen by comparing the 1-reply column to the other columns in Table II. This cost arises because there is no efficient way for the group of distributed processes to ensure that exactly one process generates a reply message. Thus the kernel receiving the replies must incur the overhead of receiving, recognizing, and discarding the second and subsequent reply packets even when only one reply is desired. It thus incurs most of the processing cost of a multiple-reply mechanism in any case.[11] Finally, the 0-reply model is provided but is not heavily used because at least one reply is required in most of our applications.

The multiple-reply model is not without disadvantages. The introduction of `GetReply` significantly complicates the kernel interface. There is a timing dependency in requesting the second and subsequent reply messages using `GetReply`. A new reply might arrive immediately after `GetReply` times out waiting for a reply message. The application must therefore estimate the time for replies to be returned, or at least the time it is willing to wait for additional replies and supply this time as a parameter to `GetReply`. (Because this is application and service dependent, the kernel cannot determine the time-out parameters.) Thus the application programmer is required to consider some of the issues that are normally handled by the transport level.

In summary, there are several possible choices for the semantics of reliable group communication and a variety of means of implementing the chosen semantics. Our design attempts to provide a simple efficient abstraction that is adequate for many applications, can be used by other applications to implement stronger semantics, and does not incur an unacceptable cost in implementing unneeded reliability. Providing multiple replies is an essential aspect of the design for a general-purpose group IPC mechanism even though this feature complicates the kernel interface and implementation.[12]

6.2 Performance

The two important measures of performance for group communication are elapsed time and processor time. Measurements of elapsed time for group IPC were presented in Section 4. Again, non-IPC group operations are less performance critical and have a cost dependent primarily on the group communication costs, and so we only consider group IPC operations. This section focuses on the

[11] Note that Table I is misleading in this respect, for it would seem from this table that there is a significant cost in receiving multiple replies. In fact, the extra cost appearing in this table is almost exclusively the result of dropped packets and associated timeouts.

[12] In fact, our original design provided only single replies for simplicity of implementation until the considerations presented above *forced* us to reevaluate the design.

cost of reliable group communication in terms of processor time and how this cost is affected by the retransmission strategy.

Processor time is important because it tends to dominate the cost of local network-based IPC [15] and because, when group IPC is used in a distributed computation, the total processor time consumed may be more indicative of the effect of group IPC on the (parallel) distributed computation than purely its elapsed time. For example, the total processor time consumed by a group Send may be more than the elapsed time because of the many concurrently executing processors involved.

To deal with processor time, we define a *packet event* as the transmission or reception of a packet. For simplicity, we assume that the processor cost of a packet event is basically the same, independent of whether it represents a transmission or reception,[13] and we ignore the effect of different lengths of packets. With the V kernel running on a SUN workstation connected to a 10-megabyte Ethernet with an efficient network interface, a packet event costs 0.4 to 1.10 milliseconds of processor time, the variance depending in part on the size of the packet.

Let N be the number of members in the group, excluding the sender. We assume that all members of the group and the sender are on separate machines. We also assume in our analysis that only machines running group members receive packets addressed to the group. Finally, we assume that the sending machine can address a packet such that it is received with high probability by all the machines running processes in the group, and no others.

6.2.1 *Cost of Group IPC Assuming No Packet Loss.* The basic cost of group communication, assuming no packet loss and using replies, is $3N + 1$ packet events: one at the sender to send a multicast packet, N on the receiving machines to each receive this packet, N on the receiving machines to transmit a reply, and N on the sending machine to receive the N replies. In contrast, simulating this with one-to-one IPC would take $4N$ packet events, with $2N$ packet events occurring on the sending machine.

These figures suggest two simple observations. First, the major performance benefit of group IPC (on a broadcast network) is in reducing the number of packet events on the sending machine by $N - 1$. This not only reduces the processor cost but also reduces the elapsed time and increases the concurrency in communication. For example, with the V kernel, the elapsed time using unreliable group IPC with $N = 10$ is roughly 3.31 milliseconds versus 32 milliseconds using one-to-one IPC for reaching all ten processes individually. Second, the cost of transmitting a multicast packet is quite high in terms of packet events generated, namely $N + 1$, and so retransmitting a multicast packet should be done only when necessary.[14]

[13] That is, the cost of fabricating the packet, transferring it to the network interface and handling the completion interrupt is comparable to the cost of a packet-received interrupt, transferring the packet from the interface and interpretation of the packet data.

[14] In reality, the cost of retransmission may be higher than that expressed here because many machines receive all multicast packets and not just packets with particular multicast addresses.

6.2.2 *Cost of Group IPC with Packet Loss.* The cost of handling packet loss is dependent on how it is handled. If retransmissions are handled by one-to-one (unicast) transmissions, the processor cost in packet events C_{pe} is given by

$$C_{pe} = 3N + 1 - 3L_s - L_r + 4(L_s + L_r),$$

where L_s is the number of receivers not receiving the multicast packet and L_r is the number of reply packets lost. (This assumes for simplicity that no retransmissions are required for the one-to-one IPC portion.) Thus the cost is basically $3N$ packet events plus four packet events for each process whose reply is not received to the group Send. As we have observed, packet loss can be expected in replies to a multicast packet if the group has a large membership or the sender's network interface has limited buffering because the multiple replies can generate numerous closely spaced packets addressed to the sending host.

An alternative retransmission scheme is to retransmit the multicast packet. The packet event cost is then roughly

$$C_{pe} = 3N + 1 + R(3N),$$

where R is the number of retransmissions. This calculation is actually high by the number of packet events missed due to packet loss. For example, if the replies from two processes are lost due to (say) network interface buffer space (and limited host processor speed in emptying the buffers), it would cost $6N$ packet events, assuming a single retransmission enabled reception of all replies. This assumption is optimistic because a retransmission is going to generate (more or less) the same set of reply packets that caused two of the original reply packets to be dropped.

This contrasts with $3N + 7$ packet events using one-to-one retransmission. Thus, if a nontrivial group is involved ($N > 2$), the one-to-one retransmission is less expensive in processor time. In fact, using N one-to-one IPC message transactions to simulate a group Send costs roughly $4N + 4R$ packet events, and so this is cheaper for nontrivial values of N as well.

In general, multicasting of retransmissions appears warranted only when elapsed time for a reliable group send is more important than total processor time consumed and loss of multiple replies is not a problem. The right choice between one-to-one retransmission and multicast retransmission (or whether retransmission is necessary at all) given these costs is application specific.

6.3 Security

The V kernel is not a *security kernel*, and so we have not implemented extensive security measures for group operations. However, we have considered some of the security issues in the design, both as a basis for design choices in the V kernel as well as to identify the issues involved in extending this design to a more secure environment. The security of non-IPC operations relies on the security of group communication plus the protection mechanisms implemented as part of each specific operation. Therefore we focus exclusively on security issues for group communication.

There are three main security issues that arise with groups:

—who may send to a group;
—who may determine or need to know the membership of a group;
—who may join a group (and therefore may reply to messages addressed to a group).

Group IPC shares with one-to-one IPC the issue of security of messages in transit.

6.3.1 *Who May Send to a Group.* An *open group* is one in which processes outside the group may send to the group. The V kernel implements open groups for several reasons. (The main alternative is to provide *closed groups* in which only members of the group may send to the group.) First, the current V kernel implementation allows a process to send to any other, and expects the receiver of messages to deal with unauthorized messages.[15] Thus restricting which processes can send to a group in the V kernel would not change which processes could send to individual processes. Second, most of the messages to a *server group* originate from clients and thus from processes outside the group. Since server groups are a common use of the V group IPC, it is necessary to provide open groups. Finally, open groups are easier to implement in the V kernel, and we suspect in other systems as well.

In general, we believe that the control over who can send to a group should match the control over who can send to an individual process or port. The V kernel imposes no such controls in either case. In systems such as DEMOS [2] or Accent [29], which provide communication security controls, it would be appropriate to carry those controls over to group interprocess communication.

6.3.2 *Who May Determine the Membership of a Group.* The use of a group-id to identify a group gives no indication of which processes belong to the group. Thus a process can send to a group without knowing which processes are members of the group. This is an advantage over the explicit list approach if the membership of the group is considered sensitive.

The other side of the issue is whether a process can determine the membership of the group. A process could discover membership information by looking at the identity of the processes that reply to a group message. To handle this case, we allow an *anonymous* reply to a group message, whereby the value returned by `Send` (normally the process id of the replying process) is the group-id to which the message was sent. Using this mechanism, a process can receive, forward, or reply to a group message and remain anonymous to the sender. However, the `QueryGroup` operation currently allows one to confirm whether or not a given process is in a group. In some circumstances, this might be regarded as a security violation.

6.3.3 *Who May Join a Group.* Control on group membership is a significant security issue. Note that with the conventional V kernel message semantics, only

[15] Note that because reply messages are tied to send operations, most applications do not receive messages in this sense. That is, reply messages can only come from the process to which the message was sent.

the receivers of a message may reply to it. Using these messages semantics, control over who may reply to a message is exercised by controlling who may join a group. If any unauthorized process is allowed to join a group, it may both receive and respond to messages in such a way as to disrupt the operation of the system. For example, suppose that an intruder joined the distributed computation and proceeded to reply to group messages with misleading information. It could cause the program to behave incorrectly as well as possibly violate the security of the system by handling secure program data. Similarly, a process might join one of the standard system server groups and respond erroneously.

In the V kernel, a process is allowed to join a restricted group if it is associated with the same user number as the other members of the group (or there are not yet any members in the group). Thus a restricted group represents a single *principal* in the security sense. An alternative approach might be called the *self-regulating group.* This would require that membership be approved by the current members of the group. A request to join a group would produce a message to the current group and require a response from application-level code. Group members would respond, indicating whether the new member is accepted or not. A variety of policies could be implemented by the group members. This mechanism appears quite easy to implement as an extension of our current design. However, we have not encountered a need for this more flexible level of group membership control. In fact, it is rare that a process discriminates between other processes other than on the basis of their associated user number.

Unrestricted groups place no restrictions on membership (except that a local unrestricted group can only have local processes as members). Unrestricted groups are primarily intended for use with queries in which security is not a problem, or the security problem can be solved at the sending end. For instance, one might query a server group to see which servers have an object with a particular name. The requesting process can then use the process identifiers and the associated user numbers of the replying processes as criteria for filtering out unwanted replies. In general, in a distributed system such as V, where a variety of servers extend the object and name space, it is useful to have unrestricted groups to allow maximum participation of servers and apply filtering on the replies at the client end.

6.3.4 *Encryption of Group Messages.* Encrypted group messages can be provided using a mechanism similar to that described by Birrell [6], provided that a process group is regarded as representing one *principal.* Such a group would have its own secret key, presumably in addition to the private (secret) key of each process in the group. Messages sent to the group would be encrypted with a conversation key as with one-to-one communication and the request for authentication and secure reply would follow similarly. The authentication server could be charged with generating secret keys for new groups and authorizing processes to join groups (by giving them the group's secret key). By using a public key encryption system, there would be a public key for each group, with the "secret" decrypt key known to all the members of the group.

In summary, group IPC raises some additional security issues. We argue that sending to a group should be controlled by the same mechanism controlling sending to an individual process. Control of group membership is important to avoid unauthorized participation in a group and is provided in V with restricted groups. If the membership of the group is considered to be sensitive, group members can return anonymous replies to group messages. Finally, existing encrypting techniques appear to carry over directly to prevent unauthorized readers and forgers of messages.

6.4 Local Groups

Local groups are provided as an optimization for the circumstances in which all the processes in a group are local to one host. For example, a single job often executes on a single host. Similarly, a parallel computation running on a multiprocessor machine would execute on what constitutes a single logical host. With a local group, one can avoid transmitting multicast packets to communicate with the group. If the sender is local to the host of the local group, no network traffic is required. If the sender is not local to the same host, a directed or unicast packet is sent to the host of the local group. Other than that, the kernel mechanisms for dealing with local groups are identical to those used for global groups: The kernel simply performs the requested operation repeatedly on each member of the local group. Thus the implementation of local groups primarily requires checking group-ids for the "local" bit and routing the packets and messages accordingly.

7. RELATED WORK

V Process groups bear some similarity to UNIX process groups [30], which are designed primarily for job control. However, a UNIX process can only belong to one process group, process group operations are limited to suspending, resuming, interrupting, and terminating the process group, and a process group is limited to a single UNIX host. There have also been many papers describing group communication mechanisms, several of which have appeared simultaneously with our original design proposal [16]. For example, Frank et al. [21] describe an extension of XNS to support multicast with considerable emphasis on routing. Cocanet [31] proposes a multicast extension to UNIX, aimed primarily at supporting distributed databases. Also, several language extensions to support multicast have been proposed [22, 25].

The use of process group facilities is similar to that described for the broadcast and multicast capabilities of local networks [7]. Similar uses and techniques were developed in Project Universe [33] using a satellite broadcast channel.

In addition, several recent papers [1, 9, 27] consider reliable broadcast protocols. Our example of reliable group communication using a logging process is a simplified version of the reliable broadcast protocol described by Chang and Maxemchuck [9]. These protocols can be implemented using our group IPC, but are only needed when the application requires additional reliability over that provided by the V kernel.

Most other work in the field focuses on the provision of broadcast and multicast at the data link and network levels. In particular, the Ethernet broadcast network requires only multicast addressing and filtering in the hosts to provide multicast delivery [26]. Implementing broadcast and multicast on point-to-point networks or internetworks has been covered by a number of authors, including Dalal and Metcalfe [19], Wall [32], and Boggs[7]. All this work addresses the efficient multicast delivery of packets to hosts, and thereby provides the necessary underlying mechanism for host groups and, therefore, for our process group facilities.

8. CONCLUDING REMARKS

We have described an extension of the V kernel to support process groups. This facility has been implemented in the V kernel and is in use by several server groups, application programs as well as the kernel itself. On the basis of our experience with this implementation and the initial applications, we conclude that the design allows a relatively simple, efficient implementation that is useful in distributed operating systems and distributed programs. In particular, we have described the use of V group IPC by several server groups, as well as by two distributed programs.

In the overall discussion, we have tried to explore issues of group communication in general, independent of the V IPC design. It appears reasonable to extend other message systems to support group operations in a fashion similar to V. It is less clear whether remote procedure call mechanisms are amenable to group communication without seriously straining normal procedure call semantics.

Despite the success of V process groups, there are a number of unresolved issues in the way groups extend the semantics of interprocess communication. First, the use of strict request–response interaction in the basic V IPC allows one to view a message as transferring control to the receiver and the reply message as returning control, similar to the behavior of a remote procedure call. This transfer of control is used by the receiver to access the sender's address space and to schedule the continued execution of the sender. The control aspect is significantly complicated by group IPC because a receiver may receive a (group) message after another receiver has replied to the sender. In this case, the second receiver has no control over the sender. Thus the message behaves more like that in conventional message systems in which it represents a transfer of data but not control.

The multiple, replicated messages generated by a group `Send` operation lead to further semantic issues. In particular, a process can receive the message associated with a particular (group) message transaction after that message transaction has been terminated by the sending process. For instance, the sender may have got a reply from another member of the group to which it sent and then performed another `Send` operation. Note that the "age" of such an old message is bounded by a time-out mechanism on queued messages. In this situation, the reply generated by the receiver will be discarded because of its old transaction identifier, and so this situation is not a problem in practice. However, in spite of these issues we have raised, the message transaction model is efficient for the one-to-one case and provides automatic discarding of "old" replies with the group IPC. In general, our practical experience with V IPC extended to group communication is very encouraging.

The group IPC has raised a practical problem with our current workstation network interfaces, namely, packet loss. As described in Section 4, most of our workstations have Ethernet interferes with only two receive buffers. Since a group `Send` operation often generates many closely spaced reply packets, a significant number of the reply packets are lost. We are looking to upgrade our workstations to Ethernet interfaces with more buffering, the obvious and seemingly only solution to this problem.

Our ongoing work is focusing on using process groups in several ways, including distributed name mapping using server groups, distributed virtual memory, parallel distributed computation, and replicated message transactions for robust operation (similar to Cooper's replicated procedure calls [18]). In general, our experience to date suggests that process group support should be considered a standard facility to be provided in distributed operating systems. Its use at the applications level will increase with the availability of this facility and the development of robust and sophisticated distributed programs.

ACKNOWLEDGMENTS

The V kernel and the V distributed system that uses it were developed by the Distributed Systems Group at Stanford. A number of other members of this group contributed to the refinement of these ideas, the use of the group IPC, and the improvement of the paper. We are particularly grateful to Michael Stumm, Tim Mann, Eric Berglund, Lance Berc, Marvin Theimer, and Ross Finlayson, as well as Stella Atkins of the University of British Columbia. We are also grateful to the referees for their helpful comments and suggestions.

REFERENCES

1. AWERBUCH, B., AND EVEN, S. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Aug. 27-29). ACM, New York, 1984, pp 278-281.
2. BASKETT, F., HOWARD, J. H., AND MONTAGUE, J. T. Task communication in DEMOS. In *Proceedings of the 6th ACM Symposium on Operating System Principles* (West Lafayette, Ind., Nov. 16-18). ACM, New York, 1977, pp. 23-31. Also published in *Oper. Syst. Rev. 11,* 5 (1977).
3. BECHTOLSHEIM, A., BASKETT, F., AND PRATT, V. The SUN workstation architecture. Tech. Rep. 229, Computer Systems Laboratory, Stanford University, Mar. 1982.
4. BERGLUND, E. J., AND CHERITON, D. R. Amaze: A distributed multi-player game program using the distributed V kernel. In *Proceedings of the 4th International Conference on Distributed Systems* (San Francisco, Calif., May 14-18). IEEE, New York, 1984, pp. 248-255.
5. BERGLUND, E. J., BROOKS, K. P., CHERITON, D. R., KAELBLING, D. R., LANTZ, K. A., MANN, T. P., NAGLER, R. J., NOWICKI, W. I., THEIMER, M. M., AND ZWAENEPOEL, W. V-System Reference Manual. Computer Science Dept. Stanford University, Jan. 1985.
6. BIRRELL, A. D. Secure communication using remote procedure calls. *ACM Trans. Comput. Syst. 3,* 1 (Feb. 1985) 1-15.
7. BOGGS, D. R. Internet broadcasting. Ph.D. dissertation, Electrical Engineering Dept., Stanford University, Oct. 1983. Also Tech. Rep. CSL-83-3, Xerox PARC, Palo Alto, Calif.
8. BOGGS, D. R., SHOCH, J. F., TAFT, E. A., AND METCALFE, R. M. PUP: An internetwork architecture. *IEEE Trans. Commun. COM-28,* 4 (Apr. 1980), 612-624.
9. CHANG, J. M., AND MAXEMCHUCK, N. F. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2,* 3 (Aug. 1984), 251-273.
10. CHERITON, D. R. *The Thoth System: Multi-Process Structuring and Portability.* Elsevier/North-Holland, New York, 1982.
11. CHERITON, D. R. The V kernel: A software base for distributed systems." *IEEE Software 1,* 2 (1984), 19-43.
12. CHERITON, D. R. An experiment in register-based interprocess communication for fast message-passing. *Oper. Syst. Rev. 18,* 4 (Oct. 1984), 12-19.
13. CHERITON, D. R. A uniform I/O interface and protocol for distributed systems. *ACM Trans. Comput. Syst.* (1985), submitted for publication.

14. CHERITON, D. R., AND MANN, T. P. Uniform access to distributed name interpretation. In *Proceedings of the 4th International Conference on Distributed Systems* (San Francisco, Calif., May 14–18). IEEE New York, 1984, pp. 290–297.
15. CHERITON, D. R., AND ZWAENEPOEL, W. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating System Principles.* (Bretton Woods, N.H., Oct. 10–13). ACM, New York, 1983, pp. 129–139.
16. CHERITON, D. R., AND ZWAENEPOEL, W. One-to-many interprocess communication in the V-System. In *Proceedings of the ACM SIGCOMM '84 Symposium on Communications Architectures and Protocols* (Montreal, Quebec; June 6–8). ACM, New York, p. 64.
17. CHERITON, D. R., MALCOLM, M. A., MELEN, L. S., AND SAGER, G. R. Thoth, a portable real-time operating system. *Commun. ACM 22,* 2 (Feb. 1979), 105–115.
18. COOPER, E. C. Replicated procedure call. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Aug. 27–29). ACM, New York, 1984, pp. 220–232.
19. DALAL, Y. K., AND METCALFE, R. M. Reverse path forwarding of broadcast packets. *Commun. ACM 21,* 12 (Dec. 1978), 1040–1048.
20. DIGITAL EQUIPMENT CORPORATION, INTEL CORPORATION, AND XEROX CORPORATION. The Ethernet: A local area network—Data link layer and physical layer specifications, Version 2.0.
21. FRANK, A., WITTIE, L., AND BERNSTEIN, A. Group communication in NetComputers. In *Proceedings of the 4th International Conference on Distributed Computing Systems* (San Francisco, Calif., May 14–18). IEEE, New York, 1984, pp. 326–335.
22. GEHANI, N. H. Broadcasting sequential processes (BSP). *IEEE Trans. Softw. Eng. SE-10,* 4 (July 1984), 343–351.
23. LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst. 4,* 3 (July 1982), 382–401.
24. LAMPSON, B. W. Atomic transactions. In *Distributed Systems: Architecture and Implementation,* B. W. Lampson, Ed. Lecture Notes in Computer Science, Springer-Verlag, New York, 1981.
25. LEBLANC, T. J., AND COOK, R. P. Broadcast communication in StarMod. In *Proceedings of the 4th International Conference on Distributed Computing Systems* (San Francisco, Calif., May 14–18). IEEE, New York, 1984, pp. 319–325.
26. METCALFE, R. M., AND BOGGS, D. R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19,* 7 (July 1976), 395–404.
27. MOCKAPETRIS, P. V. Analysis of reliable multicast algorithms for local networks. In *Proceedings of the 8th Data Communications Symposium* (North Falmouth, Mass., Oct. 3–6). ACM, New York, 1983, pp. 150–157.
28. POWELL, M. L., AND PRESOTTO, D. L. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 10–13). ACM, New York, 1983, pp. 100–109. Also published in *Oper. Syst. Rev. 17,* 5 (Oct. 1983).
29. RASHID, R., AND ROBERTSON, G. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Dec. 14–16, Pacific Grove, CA) ACM, New York, 1981, pp. 64–75.
30. RITCHIE, D. M., AND THOMPSON, K. The UNIX Time-Sharing System. *Commun. ACM 17,* 7 (July 1974), 365–375.
31. ROWE, L. A., AND BIRMAN, K. P. A local network based on the UNIX operating system. *IEEE Trans. Softw. Eng. SE-8,* 2 (Mar. 1982), 137–146.
32. WALL, D. W. Mechanisms for broadcast and selective broadcast. Ph.D. dissertation, Electrical Engineering Dept., Stanford University, June, 1980.
33. WATERS, A. G., ADAMS, C. J., LESLIE, I. M., AND NEEDHAM, R. M. The use of broadcast techniques on the UNIVERSE network. In *Proceedings of ACM SIGCOMM '84 Symposium on Communications Architectures and Protocols,* (Montreal, Quebec, June 6–8). ACM, New York, 1984, pp. 52–57.

Received February 1984; revised January 1985; accepted January 1985.