GROUP COMMUNICATION IN THE AMOEBA DISTRIBUTED OPERATING SYSTEM

M. Frans Kaashoek *Andrew S. Tanenbaum*

Dept. of Mathematics and Computer Science
Vrije Universiteit, Amsterdam, The Netherlands
(email: kaashoek@cs.vu.nl, ast@cs.vu.nl)

ABSTRACT

Many applications can profit from broadcast communication, but few operating systems provide primitives that make broadcast communication available to user applications. In this paper we introduce primitives for broadcast communication that have been integrated with the Amoeba distributed operating system. The semantics of the broadcast primitives are simple and easy to understand, but are still powerful. Our primitives, for example, guarantee global ordering of broadcast messages. The proposed primitives are also efficient: a reliable broadcast can be done in just slightly more than two messages, so, the performance is comparable to a remote procedure call. In addition, the primitives are flexible: user applications can, for example, trade performance against fault-tolerance.

1. INTRODUCTION

Most current distributed operating systems are based on remote procedure call (RPC) [4]. For many distributed and parallel applications, however, this sender-to-receiver-and-back communication style is inappropriate. What is frequently needed is *broadcasting*, in which an arbitrary one of the n user processes sends a message to the other $n - 1$ processes. Although broadcasting can always be simulated by sending $n - 1$ messages and waiting for the $n - 1$ acknowledgements, this algorithm is slow, inefficient, and wasteful of network bandwidth. In this paper we describe the broadcast primitives and their protocol for the Amoeba distributed operating system.

As an example of applications that can profit from broadcasting are those applications that replicate data structures at multiple sites. Parallel applications, for example, may want to replicate data to improve performance. Other applications may want to replicate data to enhance fault tolerance. When a process needs to change a distributed data structure, it must update or invalidate all other copies. Here, a reliable broadcast from one process to all the other processes is a more appropriate model than RPC. Many other distributed applications in which some kind of global state is needed are also candidates for using broadcasting.

Interestingly enough, broadcast communication is provided by many kinds of networks, including LANs, geosynchronous satellites, and cellular radio systems [14]. Thus, the hardware often supports the broadcasting that the applications need. It is the operating system that gets in the way. Indeed, although much research has been done in broadcast communication [6], only a few systems make the broadcast capability of the network available to user applications: V [7] and ISIS [3] are the best-known examples. In these systems a process can send a broadcast message to a group of processes. This technique is often called *group communication*.

To understand the differences between these systems, three properties of group communication are of interest: reliability, ordering, and performance. In the V system, broadcast messages are unordered. If two processes broadcast two messages, A and B, simultaneously, some of the members may receive A first and others will receive B first. Reliability in the V system means that at least one of the members must have received the message. A more reliable primitive can be built by waiting for a reply from all members. However, this is a very inefficient way of doing reliable broadcasting.

In the ISIS system, messages are globally ordered, even for groups that overlap. If, for example, processes P_1 and P_2 in Fig. 1, simultaneously send a message, processes P_3 and P_4 will receive both messages in the same order. Reliability in ISIS means that either *all* or *no* members of a group will receive a message, even in the face of processor failures. Because these semantics are hard to implement efficiently, ISIS also provides primitives that give weaker ordering, but better performance. It is up to the user application to decide which primitive is required.

In this paper, we propose a new group abstraction. Our protocol for reliable group communication is very efficient: a reliable broadcast requires only a fraction more than two messages on the average. As a result performance is comparable to RPC. The semantics of the group primitives are simple and easy to understand, but are powerful nevertheless. Our primitives, for example, guarantee per group a global ordering of broadcast messages. In addition, user applications can specify if they also want reliable communication in the presence of processor failures. A prototype implementation of the proposed group communication has been built into the Amoeba operating

CH2996-7/91/0000/0222$01.00 © 1991 IEEE

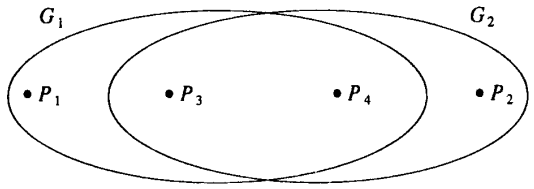

Fig. 1 Global ordering with overlapping groups. P_1 belongs to group G_1. P_2 belongs to group G_2. P_3 and P_4 are member of both groups.

system. This prototype has been used for running parallel applications [1, 2].

The outline of the rest of the paper is as follows. In Section 2, we will give a short description of Amoeba. In Section 3, we will introduce the primitives for group communication. In Section 4, we will describe the protocol needed to implement these primitives. In Section 5, we will discuss our system and compare it to other systems. In Section 6, we will draw some conclusions.

2. AMOEBA

Amoeba is an operating system specially designed for loosely-coupled computing systems [12, 16]. The system can be viewed as a collection of objects on each of which a set of operations can be performed. The list of allowed operations is defined by the person who designs the object and who writes the code to implement it. Both hardware and software objects exist.

Associated with each object is a *capability*, a kind of ticket or key that allows the holder of the capability to perform some (not necessarily all) operations on that object. Capabilities are protected cryptographically to prevent users from tampering with them.

This object model visible to the users is implemented using remote procedure call. Associated with each object type is a server process that manages the object. When a client wants to perform an operation on an object, it sends a request message to the server that manages the object. The server is addressed by a *port* that is part of the capability [15]. The message contains the capability for the object, a specification of the operation to be performed, and any parameters the operation requires. The client then blocks. After the server has performed the operation, it sends back a reply that unblocks the client.

The combination of a request from a client to a server and a reply from a server to a client is a *remote operation*. Both the request and reply messages consist of a header and a buffer. Headers are 32 bytes, and buffers can be up to 30000 bytes. A request header contains the capability of the object to be operated on, the operation code, and a limited area (8 bytes) for parameters of the operation. For example, in a write operation on a file, the capability identifies the file, the operation code is *write*, and the parameters could specify the size of the data to be written, and the offset in the file. The request buffer contains the data to be written. A reply header contains an error code, a limited area (8 bytes) for the result of the operation, and a field that can be used to return a capability (e.g., the capability of a file to be looked up in a directory). The implementation of remote operations guarantees that they are executed at most once.

Although remote operations form a useful abstraction for communication between a client and a server, some applications consist of more than one process and have a need for a different type of communication. For example, a directory service may very well consist of a number of replicated servers to increase the availability and reliability. These servers need to communicate among each other to keep the copies of directories consistent. What is needed here is some kind of broadcast communication. In the rest of the paper we describe how this is done in Amoeba.

3. GROUP COMMUNICATION

A *group* consists of one or more processes, called *members*, typically running on different processors and cooperating to provide some service or carry out some application program. Processes may be a member of more than one group. Groups are closed, which means that only group members can send a broadcast message to the group. Processes which are not a member and which wish to communicate with a group can use remote operations (or can join the group). A process need not be aware whether a service consists of multiple servers which perhaps broadcast messages to communicate with one another, or a single server. Also, a service should not have to know if the client consists of a single process or a group of processes. This design decision is in the spirit of the object model that Amoeba provides: clients know what operations are allowed, but should not know how these operations are implemented by the service. The primitives to manage groups and to communicate within a group are given in Fig. 2. We now discuss each primitive in turn.

A group is created by calling *CreateGroup*. The first parameter is a port identifying the group. The second parameter is the number of member crashes the group must survive (0 if no fault tolerance is required). This is called the *resilience degree* of a group. The other parameters of *CreateGroup* specify information that eases the implementation of a group: the maximum number of members, the number of buffers to use, and the maximum size of a message. Using this information the kernel allocates memory for buffering messages and for member information. If not enough memory is available, *CreateGroup* fails. Otherwise, it succeeds and returns a small integer, called a group descriptor, *gd*, which can be used as a quick entry point for subsequent group calls. A group descriptor is similar to a UNIX† file descriptor. Although these parameters could easily be replaced by default values, we decided against this for the sake of flexibility.

† UNIX is a Registered Trademark of AT&T Bell Laboratories.

CreateGroup(port, resilience, max_group_size, nr_buf, max_msg_size) → gd	Create a group. A process specifies how many members failures must be tolerated without loss of any message.
JoinGroup(hdr, max_group_size, nr_buf, max_msg_size) → gd	Make a process member.
LeaveGroup(gd, hdr)	Leave a group. The last member leaving causes the group to vanish.
SendToGroup(gd, hdr, buf, bufsize)	Atomically send a message to all the members of the group. All messages are globally ordered.
ReceiveFromGroup(gd, &hdr, &buf, bufsize, &more) → size	Block until a message arrives. *More* tells if the system has buffered any other messages.
ResetGroup(gd, hdr, nr_members) → group_size	Recover from processor failure. If the new group has at least *nr_member* members, it succeeds.

Fig 2. Primitives to manage a group and to communicate within a group. A message consists of a header and a buffer. An output parameter is marked with "&".

Once a group has been created, other processes can become members of it by executing *JoinGroup*. Only a member can receive messages that are sent to its group. Like *CreateGroup*, *JoinGroup* returns a group descriptor for use in subsequent group calls. In addition to adding a process to a group, *JoinGroup* delivers *hdr*, a small message, to all other members. *hdr* is an Amoeba header (described in the previous section). It contains, for example, the group's port. In this way, other members can find out that a new member has joined the group.

Once a process is member of a group, it can leave the group by calling *LeaveGroup*. When a member has left the group, it will not receive subsequent broadcasts. In addition to leaving the group, *LeaveGroup* delivers *hdr* to all other members. In this way, other members can find out that a member has left. The last member calling *LeaveGroup* causes the group to cease to exist.

When a member wants to broadcast a message, it calls *SendToGroup*. It guarantees that *hdr* and *buf* will be delivered to all members, even in the face of unreliable communication and finite buffers. Furthermore, when the resilience degree of the group is k, the protocol guarantees that in the event of a simultaneous crash of up to k members, it delivers the message to all remaining members or to none. Choosing a large value for k provides a high degree of fault-tolerance, but extracts a penalty in performance. The trade-off chosen is up to the user.

In addition to reliability, the protocol guarantees that messages are delivered in the same order to all members. Thus, if two members (on two different machines), simultaneously broadcast two messages, A and B, the protocol guarantees that either

(1) all members receives A first and then B, or
(2) all members receives B first and then A.

Random mixtures, where some members get A first and others get B first are guaranteed not to occur. By making the group primitives both reliable and indivisible in this way, the user semantics become much simpler and easier to understand.

To receive a broadcast, a member must call *ReceiveFromGroup*. If a broadcast arrives and no such primitive is outstanding, the message is buffered. When the member finally does a *ReceiveFromGroup*, it will get the next one in sequence. How this is implemented will be described below. The *more* flag is used to indicate to the caller that one or more broadcasts have been buffered and can be fetched using *ReceiveFromGroup*. If a member never calls *ReceiveFromGroup*, the group may block, because it may run out of buffers. Messages are never intentionally thrown away.

The final primitive allows recovery from member crashes. If one of the members (or its kernel) does not respond to messages, the protocol enters a recovery mode. All outstanding calls return an error value indicating that a member has crashed. The user application can now call *ResetGroup* to transform the group into a new group that contains as many alive members from the group as possible. The second parameter is the number of members that the new group must contain as a minimum. When *ResetGroup* succeeds, it returns the group size of the new group. In addition to recovering from crashes, *ResetGroup* delivers *hdr* at all new members. It may happen that multiple members initiate a recovery at the same moment. The new group consisting of the members that can communicate with each other, however, is only built once.

The way recovery is done is based on the design principle that policy and mechanism should be separated. In many systems that deal with fault tolerance, recovery from processor crashes is completely invisible to the user application. We decided not to do this. A parallel application that multiplies two matrices, for example, may want to continue even if only one processor is left. A banking system may require, however, that at least half of the group is alive. In our system, the user is able to decide on the policy. The group primitives only provide the mechanism.

To summarize, the group primitives provide an abstraction that enables programmers to design applications that consist of one or more processes that run on different machines. It is a simple, but powerful abstraction. All members of a group see all events in the same order. Even the events of a new member joining the group, a member leaving the group, and recovery from a crashed member are globally ordered. If, for example,

one process calls *JoinGroup* and a member calls *SendToGroup*, all members first receive the join and then the broadcast or first receive the broadcast and then the join. In the first case the process that called *JoinGroup* will also receive the broadcast message. In the second case, it will not receive the broadcast message. A mixture of these two events is guaranteed not to happen. This property makes reasoning about a distributed application much easier. Furthermore, the group interface gives support for building fault tolerant applications by choosing an appropriate resilience degree.

4. THE PROTOCOL FOR GROUP COMMUNICATION

The protocol to be described runs inside the kernel. It assumes that unreliable message passing between entities is possible; fragmentation, reassembly, and routing of messages is done at lower layers in the kernel. If the underlying network does not have a broadcast or multicast capability, the kernel will simulate this by making copies of messages and sending them point-to-point. This will hurt the performance of the protocol considerable, so we assume in our description that a network supports broadcast or multicast.

Each kernel running a member keeps information about the group (or groups) to which the member belongs. It stores, for example, the size of the group and information about the other members in the group. Any member of a group can, at any instant, decide to send a broadcast message to its group. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communications, lost packets, finite buffers, and node failures. We assume, however, that byzantine failures (a process sends spurious or contradictory messages) do not occur.

Without loss of generality, we assume that the system contains one group with each member running on a separate processor (see Fig. 3). The hardware of all machines is identical, and they run exactly the same kernel and application software. However, when the application starts up, one of the machines is elected as *sequencer*. The machine on which the group is created is made the sequencer. If the sequencer machine subsequently crashes, the remaining members elect a new one (described below).

4.1. The Protocol for Communication Failures

The basic protocol works as follows (an elaborate description is given in [9]). When a group member calls *SendToGroup* to send a message, M, it hands the message to its kernel and is blocked. The kernel encapsulates M in an ordinary point-to-point message and sends it to the sequencer. The sequencer contains the same code as all other members. The only difference is that a flag tells it to process messages differently.

When the sequencer receives the point-to-point message containing M, it allocates the next sequence number, s, and broadcasts a packet containing M and s. Thus all broadcasts are issued from the same node, the sequencer. Assuming that no packets are lost, it is easy to see that if two members

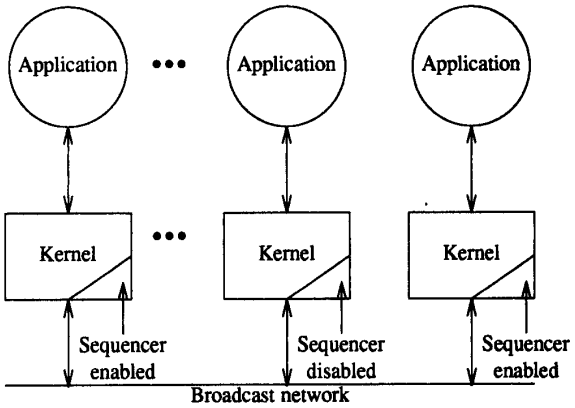

Fig 3. System structure. Each node runs a kernel and a user application. Each kernel is capable of being sequencer, but, at any instant, only one of them functions as sequencer. If the sequencer crashes the remaining nodes can elect a new one.

simultaneously want to broadcast, one of them will reach the sequencer first and its message will be broadcast to all the other nodes first. Only when that broadcast has been completed will the other broadcast be started. The sequencer provides a global ordering in time. In this way, we can easily guarantee per group the indivisibility of broadcasting.

When the kernel that has sent M, itself receives the message from the network, it knows that its broadcast has been successful. It unblocks the member that called *SendToGroup*.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast packet eventually arrives, the kernel will immediately notice a gap in the sequence numbers. If it was expecting s next, and it got $s+1$, it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for copies of the missing message (or messages, if several have been missed). To be able to reply to such requests the sequencer stores broadcast messages in the *history buffer*. The sequencer sends the missing messages to the process requesting them as point-to-point messages. The other members also keep a history buffer, to be able to recover from sequencer failures (as we will see in the next section) and to buffer messages if there is not outstanding a *ReceiveFromGroup* call.

As a practical matter, a kernel has only a finite amount of space in its history buffer, so it cannot store broadcast messages forever. However, if it could somehow discover that all members have received broadcasts up to and including k, it could then purge the first k broadcast messages from the history buffer.

The protocol has several ways of letting a kernel discover this information. For one thing, each point-to-point message to

the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message. This information is also included in the message from the sequencer to the other kernels. In this way, a kernel can maintain a table, indexed by member number, showing that member i has received all broadcast messages 0 up to T_i, and perhaps more. At any instant, a kernel can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, a kernel knows that everyone has received broadcasts 0 through 6, so they can be safely deleted from the history buffer.

If a node does not do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval, Δt, send the sequencer a special packet acknowledging all received broadcasts. The sequencer can also request this information when it runs out of space in its history buffer.

In short, to do a broadcast, an application process sends the data to the sequencer, which gives it a sequence number and broadcasts it. In general, no separate acknowledgement packets are used, but all messages to the sequencer carry piggybacked acknowledgements. When a node receives an out-of-sequence broadcast, it buffers the broadcast temporarily, and asks the sequencer for the missing broadcasts. Since members are cooperating on the same task, broadcasts are expected to be common—many per second—and therefore the only effect that a missed broadcast has is causing some application process to get behind by a few tens of milliseconds once in a while, hardly a serious problem. So, in general, a reliable broadcast costs only 2 messages, one point-to-point and one broadcast.

There is a subtle design point concerning the performance of the protocol. There are two ways to do a broadcast. In method 1, which we have just described, the sender sends a point-to-point message to the sequencer, which then broadcasts it. In method 2, the sender broadcasts the message. When the sequencer sees the broadcast, it broadcasts a special *accept* message containing the newly assigned sequence number. A broadcast message is only "official" when the *accept* message has been sent.

These methods are logically equivalent, but they have different performance characteristics. In method 1 each message appears on the network twice: once to the sequencer and once from the sequencer. Thus a message of length n bytes consumes $2n$ bytes of network bandwidth. However, only the second message is broadcast, so each user machine is interrupted only once (for the second message).

In method 2 the full message appears only once on the network, plus a very short *accept* message from the sequencer. Thus, only about n bytes of bandwidth are consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *accept*. Thus method 1 wastes bandwidth to reduce interrupts and method 2 minimizes bandwidth usage at the cost of more interrupts. In our current implementation we use method 1 for small messages and method 2 for large messages.

4.2. The Protocol for Processor Failures

Processor failures are detected by a kernel when it tries to reach another kernel to deliver a message. If, after a certain number of trials, a kernel has not responded, the member running on that kernel is assumed to be dead. In reality a kernel may be busy doing something else and respond a week later. This does not affect, however, the correct functioning of the group primitives.

After a member failure is detected, all further group primitives return an error status. Any surviving member may call *ResetGroup* to recover from a member failure. *ResetGroup* tries to re-form the group into a group that contains the living members that can communicatie with each other. If needed, it also elects a new sequencer. The second parameter of *ResetGroup* is the minimum number of members of the old group that are required for the new group to be valid. If *ResetGroup* succeeds, it returns the actual number of members in the new group. *ResetGroup* fails if it cannot form a group with enough members.

The protocol to recover from member crashes resembles the invitation protocol described in [8]. It runs in two phases. In the first phase, the protocol establishes which members are alive and chooses one member as coordinator to handle the second phase. Every member that calls *ResetGroup* initially becomes a coordinator and invites other members to join the new group. If a member is alive and it is not a coordinator, it responds with the highest sequence number that it has seen (each member already keeps this number for running the protocol for communication failures as described above). If one coordinator invites another coordinator, the one with the highest sequence number becomes coordinator of both (if their sequence numbers equal, the one with the lowest member id is chosen). When all members of the old group have been invited, there is one coordinator left. It knows which members are alive and which member has the highest sequence number.

In the second phase of the recovery, the group is restarted. If the coordinator has missed some messages, it asks the member with the highest sequence number for retransmissions (this is unlikely to happen, because the initiator of the recovery with the highest sequence number becomes coordinator). Now the coordinator is up-to-date, it builds a *result* message, containing information about the new group: the size of the new group, the members in the new group, the new sequencer (itself), and the new *incarnation* number of the group. (The incarnation number is included to make sure that messages directed to the old group will not be accepted by the new group.) It stores this message in its history and broadcasts it to all members. When a member receives the *result* message, it checks for missing messages. After collecting these from the coordinator, it updates the group information, sends an acknowledgement to the coordinator, and enters normal operation. The coordinator enters normal operation after it has received an acknowledgement for the *result* message from all members.

When back in normal operation, members never accept messages from a previous incarnation of the group. Thus, members that have been quiet for a long time and did not take part in the recovery will still use an old incarnation number when sending a message to the new group. These message will

be ignored by the new group, treating the ex-member effectively as a dead member. The incarnation numbers make sure that no conflicts will arise when a member suddenly comes to alive after being quiet for a period of time.

If the coordinator (or one of the members) crashes during the recovery, the protocol starts again with phase 1. This continues until the recovery is successful or until there are not enough living members left to recover successfully.

In Fig. 4 the recovery protocol is illustrated. In Fig. 4(a) a possible start of phase 1 is depicted. Members 0, 1, and 2 start simultaneously the recovery and are coordinators. Member 3 has received more messages (it has seen the highest sequence number), but it did not call *ResetGroup*. Member 4, the sequencer, has crashed. In Fig. 4(b), the end of phase 1 is reached. Member 0 is the coordinator and the other members are waiting for the *result* message (they check periodically if member 0 is still alive). In Fig. 4(c), the end of phase 2 is reached. Member 0 is the new sequencer. It has collected message 34 from member 3 and has stored the *result* message (number 35) in its history. The other members are also back in normal operation. They have collected missing messages from member 0 and have also received the *result* message.

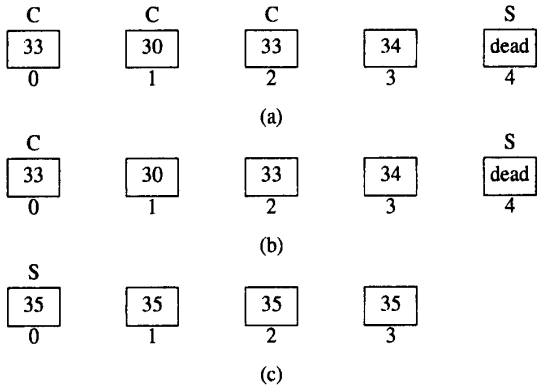

Fig. 4 A possible recovery for a group of size 5 after a crash of member 4. S is a sequencer. C is a coordinator. The number in the box is the sequence number. The number below the box is the member id. (a) Shows a possible start of phase 1. (b) Shows the start of phase 2. In (c) the recovery is completed.

The protocol described so far recovers from member failures, but does not guarantee that all surviving members receive all messages that have been sent. For example, a process sends a message to the sequencer, which broadcast it. The sender receives the broadcast and returns control to the application, which interacts with the external world. Unfortunately, all other processes miss the broadcast, and the sender and sequencer both crash.

To avoid this situation, *CreateGroup* has a parameter *resilience* that specifies the resiliency. This means that the the *SendToGroup* primitive does not return control to the application until the kernel knows that at least other *resilience* kernels have the received the message. To achieve this the kernel broadcasts the message using method 2, so that all kernels will receive the message. The sequencer buffers the message until it receives from the *resilience* lowest-numbered kernels acknowledgement messages. After receiving the acknowledgments, the sequencer allocates the next sequence number and broadcasts the *accept* message. On receiving the *accept* message, *resilience+1* kernels store the broadcast message in their history buffer. That way, no matter which *resilience* machines crash, there will be at least one left containing the full history, so everyone else can be brought up to date after the recovery. Thus, an increase in fault-tolerance is paid for by a decrease in performance. The trade-off chosen is up to the user.

5. DISCUSSION

In this section, we will compare our reliable broadcast protocol with other protocols and our system with other systems that provide broadcast communication. Fig. 5 summarizes the results. In comparing protocols, several points are of interest. First is the performance of the protocol. This has two aspects: the time before a message can be delivered to the application and the number of protocol packets needed to broadcast the message. Second are the semantics of sending a broadcast message, which has three aspects: reliability, ordering, and fault-tolerance. Although fault-tolerance can be considered as an aspect of reliability, we have decided to consider it as a separate aspect. Third is the hardware cost. The key aspect here is whether the protocol requires members to be equipped with additional hardware (e.g., a disk). Although more research has been done in broadcast communication than is listed in the table, this other research focuses on different aspects (e.g, multicast routing in a network consisting of point-to-point communication links). For these papers we refer the reader to [6].

Let us look at each protocol in turn. The protocols described by [3] are implemented in the ISIS system. The ISIS system is primarily intended for doing fault-tolerant computing. Thus, ISIS tries to make broadcast as fast as possible in the context of possible processor failures. Our system is intended to do reliable ordered broadcast as fast as possible. If processor failures occur, some message may be lost. If, however, an application requires fault-tolerance our system can trade performance against fault-tolerance. As reliable ordered broadcast in the event of processor failures is quite expensive, ISIS has primitives that provide a weaker ordering (e.g., a causal ordering). These primitives have better performance, but still require more messages than our primitive for doing ordered broadcasting.

Chang and Maxemchuk describe a family of protocols [5]. The protocols differ mainly in the degree of fault-tolerance that they provide. Our protocol for resilience degree 0 resembles their protocol that is not fault-tolerant (i.e., it may lose messages if processors fail), but ours is optimized for the common case of no communication failures. Like our protocol, the CM protocol depends also on a central node, the token site, for ordering messages. However, on each acknowledgement another node takes over the role of token site. Depending on the system utilization, the transfer of the token site on each acknowledgement can take one extra control message. Thus their protocol requires 2 to 3 messages per broadcast, whereas ours requires only 2 in the best case.

Protocol	Performance		Semantics			Additional Hard-ware
	Delay	# Pkts	Reliable	Ordering	Fault-tolerance	
Birman and Joseph [3]	2 Rounds	$2n$	Yes	Yes	n	No
Chang and Maxemchuk [5]	2	$2+\epsilon$	Yes	Yes	$0...n$	No
Cheriton and Zwaenepoel [7]	2	$2...n$	No...Yes	No	No	No
Luan and Gligor [10]	3 Phases	$1...4n$	Yes	Yes	Yes	Yes
Melliar-Smith et al. [11]	Random	1	Yes	Yes	$n/3$	No
Navaratnam et al. [13]	2	$n+1$	Yes	No...Yes	$0...n$	No
Tseung [17]	2	3	Yes	Yes	No...Yes	Yes
Our protocol	2	2	Yes	Yes	$0...n$	No

Fig 5. Comparison between different broadcast protocols. A protocol is identified by the name of the authors of the protocol. n is the group size. In a *round* each member sends a message. A *phase* is the time necessary to complete a state transition (sending messages, receiving messages, and local computation). For each protocol, we list the best performance. The performance may decrease, for example, for higher degrees of fault-tolerance.

Fault-tolerance is achieved in the CM protocol by transferring the token. If a message is delivered after the token has been transferred L times, then L processor failures can be tolerated. This scheme introduces a very long delay before a message can be delivered. Finally, in the CM protocol all messages are broadcast, whereas our protocol uses point-to-point messages whenever possible, reducing interrupts and context switches at each node. This is important, because the efficiency of the protocol is not only determined by the transmission time, but also (and mainly) by the processing time at the nodes. In their scheme, each broadcast causes at least $2(n-1)$ interrupts, ours only n.

The group communication described in [7] integrates RPC communication with broadcast communication in a flexible way. If a client sends a request message to a process group, V tries to deliver the message at all members in the group. If any one of the members of the group sends a reply back, the RPC successfully returns. Additional replies from other members can be collected by the client by calling *GetReply*. Thus, the V system does not provide reliable, ordered broadcasting. However, this can be implemented by a client and a server (e.g., the protocol described by Navaratnam, Chanson, and Neufeld runs on top of V). In this case, a client and a server should know how they are implemented. We do not think this is a good approach. If an unreplicated file server, for example, is re-implemented as a replicated file server to improve performance and to increase availability, it could mean that all client programs have to be changed.

The protocol described in [10] is one of two protocols that require additional hardware. In this protocol each member must be equipped with a disk. Using these disks the protocol can provide reliable ordered broadcasting, even if a network partitions. It uses a majority-consensus decision to commit on a unique ordering of broadcast messages that have been received and stored on disk. Under normal operation the protocol requires $4n$ messages. However, under heavy load the number of messages goes down to 1. The delay before a message can be delivered is constant: the protocol needs three protocol phases before it can be delivered. In a system like Amoeba that consists of a large number of processors, equipping each machine with a disk would be far too expensive. Furthermore, the performance of the protocol is also much too low to be considered as a general protocol for reliable broadcasting.

A totally different approach to reliable broadcasting is described in [11]. They describe a protocol that achieves reliable broadcast with a certain probability. If processor failures occur, it may happen that the protocol cannot decide on the order in which messages must be delivered. They claim that the probability is high enough to assume that all messages are ordered globally, but nevertheless there is a certain chance that messages are not globally ordered. The protocol uses only one message, but a message cannot be delivered at an application until several other broadcast messages have been received. For a group of 10 nodes, a message can be delivered on average after receiving another 7.5 messages. Thus with large groups, the delay is unacceptably large.

Navaratnam, Chanson, and Neufeld provide two primitives for reliable broadcasting [13]. One orders messages; the other does not. Their protocol also uses a centralized scheme, but instead of transferring the token site on each acknowledgement, their central site waits until it has received acknowledgements from *each* node that runs a member before sending the

next broadcast. In an implementation of the NCN protocol on the V-system, a reliable broadcast message costs 24.8 msec for a group of 4 nodes. Our current implementation does this in less than 2.5 msec (an earlier tuned prototype with less functionality did it in 1.4 msec). Ours is thus an order of magnitude faster.

The last protocol that provides reliable broadcasting is described in [17]. It requires that at least three components are added to a network: a Retransmission Computer, a Designated Recorder Computer, and one or more Playback Recorder Computers. The Playback Recorder Computers should be equipped with a disk (typically one Playback Recorder Computer is used per group). If fault-tolerance is required, hot backup systems can be provided for the Retransmission Computer and the Designated Recorder Computer. The protocol works as follows. A user computer sends point-to-point a message to the Retransmission Computer. The Retransmission Computer plays a similar role as our sequencer. It adds some information to the message, such as a sequence number, and broadcasts it. In Tseung's protocol, it is ready to broadcast the next message after the Designated Recorder Computer has sent an acknowledgement. The Designated Recorder stores messages for a short period, in case one of the Playback Recorder Computers has missed a message. The Playback Computers store the messages on disk for a long period of time to be able to send retransmissions to user computers if they have missed a message. This protocol requires more messages than our protocol (the acknowledgment from the Designated Recorder to the Retransmission Recorder is not needed in our protocol) and requires additional hardware. Furthermore, one computer (the Retransmission Computer) plays the sequencer for all groups. If the sequencer becomes a bottleneck in one group, all other groups will suffer from this. Also, if the Retransmission Computer or the Designated Recorder crashes no group communication can take place in the whole system. For these reasons and the fact that groups are mostly unrelated, we order messages only per group by having a sequencer per group.

If messages are sent regularly and if messages may be lost when processor failures occur, our protocol is more efficient than any of the protocols described above. In our protocol, the number of messages used is determined by the size of the history buffer and the communication pattern of the application. In the normal case, 2 messages, a point-to-point message to the sequencer and a broadcast message, are used. In the worst case, when one of the nodes is continuously broadcasting, $(n / HISTORY\_SIZE) + 2$ messages are needed. For example, if the number of buffers in the history is equal to the number of processors, 3 messages per reliable broadcast are needed. In practice, with say 1Mb history buffers and 1Kb messages, there is room for 1024 messages. This means that the history buffer will rarely fill up and the protocol will actually average 2 messages per reliable broadcast. The delay before a message can be delivered to the application is optimal; as soon as a broadcast arrives, it can be delivered. Also, our protocol causes a low number of interrupts. Each node gets one interrupt for each reliable broadcast message.

If messages must be delivered despite member crashes, the costs of the protocol increase. For a resilience degree of r, each reliable broadcast takes $2 + r$ messages. One message for the broadcast message from a member to all kernels, r short acknowledgements that are sent point-to-point to the sequencer, and one broadcast message from the sequencer to all members. The delay increases. A message can only be accepted by the sequencer directly after receiving the message and r acknowledgements. However, the r acknowledgments will be received almost simultaneously. Thus, for an increase of fault-tolerance the application pays with a decrease in performance. It is up to the user application to make the tradeoff.

Like some of other protocols, our protocol uses a centralized node (the sequencer) to determine the order of the messages. Although in our protocol this centralized node does not do anything computationally intensive (it receives a message, adds the sequence number, and broadcasts it), it could conceivably become a bottleneck in a very large group. To analyze this possibility, we have made a queuing theoretic model to see at what point the sequencer saturates. For the most broadcast-intensive applications we have tried so far, the issue begins to be a problem at around 400 nodes. For most applications, however, well over 1000 nodes can be supported easily. If the resilience degree is greater than 0, it is likely that the sequencer will become a bottleneck sooner due to the *resilience* acknowledgements that it has to process. Under heavy load, one could try to piggyback these acknowledgements on other messages, to scale the protocol for a high resilience degree also to a large number of processors.

6. CONCLUSION

We have presented a simple protocol that achieves reliable broadcast and guarantees that all messages will be received by every live node of a group in the same order. We have described how this protocol is implemented in the Amoeba operating system. We proposed a simple, but powerful interface that makes reliable ordered broadcasting available to user applications. The interface allows the user to trade performance against fault-tolerance.

ACKNOWLEDGEMENTS

We would like to thank Wiebren de Jonge for an improvement in the protocol, Hans van Staveren for his help with implementing the protocol, Henri Bal, Arnold Geels, Henk Schepers, and the anonymous referees for their critical reading of the manuscript and their useful comments.

REFERENCES

1. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., "A Distributed Implementation of the Shared Data-object Model," *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL, pp. 1-19 (Oct. 1989).
2. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., "Experience with Distributed Programming in Orca," *IEEE CS Int. Conf. on Computer Languages*, New Orleans, LA, pp. 79-89 (March 1990).

3. Birman, K.P. and Joseph, T.A., "Reliable Communication in the Presence of Failures," *ACM Trans. on Comp. Syst.* **5**(1), pp. 47-76 (Feb. 1987).

4. Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Trans. on Comp. Syst.* **2**(1), pp. 39-59 (Feb. 1984).

5. Chang, J. and Maxemchuk, N.F., "Reliable Broadcast Protocols," *ACM Trans. on Comp. Syst.* **2**(3), pp. 251-273 (Aug. 1984).

6. Chanson, S.T., Neufeld, G.W., and Liang, L., "A Bibliography on Multicast and Group Communication," *Operating Systems Review* **23**(4), pp. 20-25 (Oct. 1989).

7. Cheriton, D. and Zwaenepoel, W., "Distributed Process Groups in the V kernel," *ACM Trans. on Comp. Syst.* **3**(2), pp. 77-107 (May 1985).

8. Garcia-Molina, H., "Elections in a Distributed Computing System," *IEEE Trans. on Comp.* **31**(1), pp. 48-59 (Jan. 1982).

9. Kaashoek, M.F., Tanenbaum, A.S., Flynn Hummel, S., and Bal, H.E., "An Efficient Reliable Broadcast Protocol," *Operating Systems Review* **23**(4), pp. 5-20 (Oct. 1989).

10. Luan, S.W. and Gligor, V.D., "A Fault-tolerant Protocol for Atomic Broadcast," *IEEE Trans. on Parallel and Distributed Systems* **1**(3), pp. 271-285 (July 1990).

11. Melliar-Smith, P.M., Moser, L.E., and Agrawala, V., "Broadcast Protocols for Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems* **1**(1), pp. 17-25 (Jan. 1990).

12. Mullender, S.J., Rossum, G., Tanenbaum, A.S., Renesse, R. van, and Staveren, H. van, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer* **23**(5), pp. 44-53 (May 1990).

13. Navaratnam, S., Chanson, S., and Neufeld, G., "Reliable Group Communication in Distributed Systems," *Proc. 8th Int. Conf. on Distr. Comp. Syst.*, San Jose, CA, pp. 439-446 (June 1988).

14. Tanenbaum, A.S., *Computer Networks 2nd ed.*, Prentice/Hall, Englewood Cliffs, NJ (1989).

15. Tanenbaum, A.S., Mullender, S.J., and Renesse, R. van, "Using Sparse Capabilities in a Distributed Operating System," *Proc. 6th International Conference on Distributed Computing Systems*, Cambridge, MA, pp. 558-563 (May 1986).

16. Tanenbaum, A.S., Renesse, R. van, Staveren, H. van, Sharp, G.J., Mullender, S.J., Jansen, J., and Rossum, G. van, "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM* **33**(12) (Dec. 1990).

17. Tseung, L.N., "Guaranteed, Reliable, Secure Broadcast Networks," *IEEE Network Magazine* **3**(6), pp. 33-37 (Nov. 1989).