EFFICIENT RELIABLE GROUP COMMUNICATION FOR DISTRIBUTED SYSTEMS

M. Frans Kaashoek

M.I.T. Laboratory for Computer Science†
Cambridge, MA

Andrew S. Tanenbaum

Dept. of Math and Comp. Science
Vrije Universiteit
Amsterdam, The Netherlands

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, powerful, and easy to understand. Our primitives, for example, guarantee total ordering of broadcast messages. The proposed primitives are also efficient: if a network supports physical multicast, a reliable broadcast can be done in just slightly more than two messages on the average, so, the performance of a reliable broadcast is roughly comparable to that of a remote procedure call. In addition, the primitives are flexible: user applications can, for example, trade performance against fault tolerance.

1. Introduction

Many distributed applications are easier to build if the operating system supports a primitive that allows the application to send a message to *n* destinations. Like a primitive for point-to-communication, such a *broadcast* primitive must be able to recover from lost messages due to communication failures or buffer overflow. Furthermore, such a primitive should provide a way to deal with the case where some of the *n* destinations fail due to hardware or software errors. In this paper, we describe a set of primitives for *group communication* that provide reliable broadcast communication in the presence of communication and processor failures. These group communication primitives have been implemented in the Amoeba distributed operating system [Mullender et al. 1990; Tanenbaum et al. 1990]. We describe the algorithms and give extensive performance measurements. The

---

† This work was done as part of the author's Ph.D. thesis at the Vrije Universiteit [Kaashoek 1992].

group communication primitives have been used in running parallel applications [Bal 1990; Tanenbaum et al. 1992], a fault-tolerant implementation of the Orca programming language [Kaashoek et al. 1992], and a fault-tolerant distributed directory service [Kaashoek et al. 1993a].

In distributed applications, a group of processes cooperates to provide a single service. Most current operating systems provide only point-to-point communication for distributed applications, but what often also is needed is 1-to-*n* communication [Chang 1984; Gehani 1984; Dechter and Kleinrock 1986; Ahamad and Bernstein 1985; Cheriton and Zwaenepoel 1985; Joseph and Birman 1986; Liang et al. 1990; Cristian 1991; Birman 1993]. Consider, for example, a parallel application. Typically in a parallel application a number of processes cooperate to compute a single result. If one of the processes finds a partial result (e.g., a better bound in a parallel branch-and-bound program) it is desirable that this partial result be communicated immediately to the other processes. By receiving the partial result as soon as possible, the other processes do not waste cycles on computing something that is not interesting anymore, given the new partial result.

Now consider a second application: a fault-tolerant storage service. A reliable storage service can be built by replicating data on multiple processors each with its own disk. If a piece of data needs to be changed, the service either has to send the new data to all processes or invalidate all other copies of the changed data. If only point-to-point communication were available, then the process would have to send $n - 1$ reliable point-to-point messages. In most systems this will cost at least $2(n - 1)$ packets (one packet for the actual message and one packet for the acknowledgement). If the message sent by the server has to be fragmented into multiple network packets, then the cost will be even higher. This method is slow, inefficient, and wasteful of network bandwidth.

In addition to being expensive, building distributed applications using only point-to-point communication is often difficult. If, for example, two servers in the reliable storage service receive a request to update the same data, they need a way to consistently order the updates, otherwise the data will become inconsistent. The problem is illustrated in Figure 1. The copies of variable *X* become inconsistent because the messages from Server 1 and Server 2 are not ordered. What is needed is that all servers receive all messages in the same order.

Many network designers have realized that 1-to-*n* communication is an important tool for building distributed applications; *broadcast* communication is provided by many networks, including LANs, geosynchronous satellites, and cellular radio systems [Tanenbaum 1989]. Several commonly used LANs, such as Ethernet and some rings, even provide *multicast* communication. Using multicast communication, messages can be sent exactly to the group of machines that are interested in receiving the message. Future networks, like Gigabit LANs, are also likely to implement broadcasting and/or multicasting to support high-performance multi-media applications [Kung 1992].

The protocol presented in this paper for group communication uses the hardware multicast capability of a network, if one exists. Otherwise, it uses broadcast messages or point-to-point messages, depending on the size of the group and the availability of broadcast communication. Thus, this system makes any existing hardware support for group communication available to application programs.

The paper makes the following contributions:

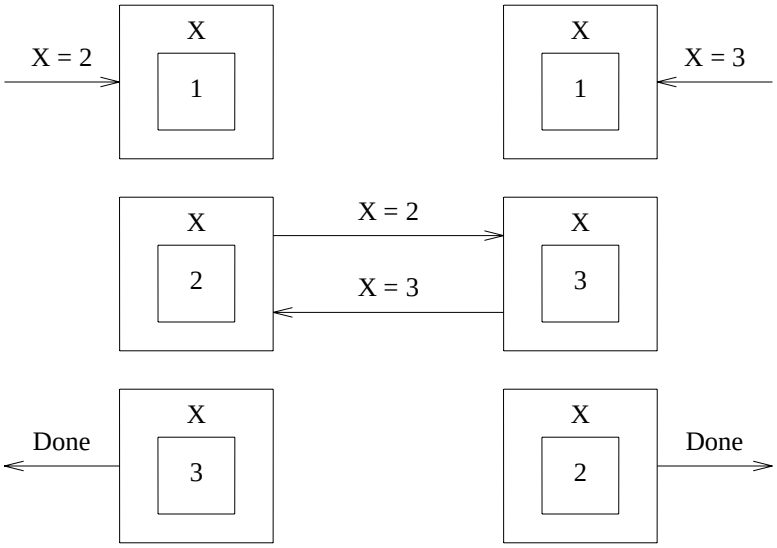


Fig. 1. Servers 1 and 2 receive update messages for the shared variable *X* at approximately the same time. The copies of *X* become inconsistent because the messages from Server 1 and Server 2 are not ordered.

- It identifies and discusses design issues in group communication.
- It introduces an improved algorithm for reliable totally-ordered group communication.
- It provides enough detail of the algorithm that it can be implemented in any distributed system.
- It gives detailed performance measurements of an implementation in an existing distributed system.

A word on terminology. The terms ‘‘broadcasting,’’ ‘‘multicasting,’’ and ‘‘group communication’’ are often confused. We will refer to the abstraction of a group of processes communicating by sending messages from 1 to *n* destinations as ‘‘group communication.’’ We consider ‘‘broadcasting’’ and ‘‘multicasting’’ as two hardware mechanisms that can be used to implement the abstraction of group communication. A broadcast message on a network is received by all processors on that network. A multicast message on a network is only received by the processors on that network that are interested in receiving it. We will often use the term ‘‘broadcasting’’ to refer to any of the three terms, as this is the term generally used in the literature.

The outline of the paper is as follows. In Section 2 we discuss the design issues in group communication. In Section 3 we present the choices that we have made for each design issue and the reasons why. In Section 4 we present the Amoeba kernel primitives for group communication. In Section 5 we give the algorithms for an efficient reliable broadcast protocol that provides total ordering.

In Section 6 we describe the structure of the Amoeba kernel and give detailed performance measurements about the group communication. In Section 7 we compare our protocol with a number of other protocols and other systems that support group communication. In Section 8 we give our conclusions.

2. Design Issues in Group Communication

A few existing operating systems provide application programs with support for group communication [Liang et al. 1990]. To understand the differences between these existing systems, six design criteria are of interest: addressing, reliability, ordering, delivery semantics, response semantics, and group structure (see Fig. 2). We will now discuss each one in turn.

Issue	Description
Addressing	Addressing method for a group (e.g., list of members)
Reliability	Reliable or unreliable communication?
Ordering	Order among messages (e.g., total ordering)
Delivery semantics	How many processes must receive the message successfully?
Response semantics	How to respond to a broadcast message?
Group structure	Semantics of a group (e.g., dynamic versus static)

Fig. 2. The main design issues for group communication.

At least four methods of *addressing* messages to a group exist. The simplest one is to require the sender to explicitly specify all the destinations to which the message should be delivered. A second method is to use a single address for the whole group. This method saves bandwidth compared to the first one and also allows a process to send a message without knowing which processes are members of the group [Frank et al. 1985]. Two less common addressing methods are *source addressing* [Gueth et al. 1985] and *functional addressing* [Hughes 1988]. Using source addressing, a process accepts a message if the source is a member of the group. Using functional addressing a process accepts a message if a user-defined function on the message evaluates to true. The disadvantage of the latter two methods is that they are hard to implement with current network interfaces.

The second design criterion, *reliability*, deals with recovering from communication failures, such as buffer overflows and garbled packets. Because reliability is more difficult to implement for group communication than for point-to-point communication, a number of existing operating systems provide *unreliable* group communication [Cheriton and Zwaenepoel 1985; Rozier et al. 1988], whereas almost all operating systems provide *reliable* point-to-point communication, for example, in the form of remote procedure call (RPC) [Birrell and Nelson 1984].

Another important design decision in group communication is the *ordering* of messages sent to

a group. Roughly speaking, there are 4 possible orderings: no ordering, FIFO ordering, causal ordering, and total ordering. Group communication without an order among messages is easy to implement in the operating system, but unfortunately makes programming for application builders harder, as they have to code up their own protocols. FIFO ordering guarantees that all messages from a member are delivered in the order in which they were sent. Causal ordering guarantees that all messages that are related are ordered [Birman et al. 1991]. More specifically: messages are in FIFO order and if a member after receiving message *A* sends a message *B*, it is guaranteed that all members will receive *A* before *B*. In the total ordering, each member receives all messages in the same order. The last ordering is stronger than any of the other orderings and makes programming easier, but it is harder to implement.

To illustrate the difference between causal and total ordering, consider a service that stores records for client processes. Assume that the service replicates the records on each server to increase availability and reliability and that it guarantees that all replicas are consistent. If a client may only update its own records, then it is sufficient that all messages from the same client will be ordered. Thus, in this case a causal ordering can be used. If a client may update any of the records, then a causal ordering is not sufficient. A total ordering on the updates, however, is sufficient to ensure consistency among the replicas. To see this, assume that two clients, C_1 and C_2, send an update for record *X* at the same time. As these two updates will be totally-ordered, all servers either (1) receive first the update from C_1 and then the update from C_2 or (2) receive first the update from C_2 and then the update from C_1. In either case, the replicas will stay consistent, because every server applies the updates in the same order. If in this case causal ordering had been used, it might have happened that the servers applied the updates in different orders, resulting in inconsistent replicas.

A debate on what the appropriate ordering is for group messages is now raging. For example, Cheriton and Skeen argue that the communication should not support any ordering at all [Cheriton and Skeen 1993]. Van Renesse, on the other hand, explicitly argues for ordering of group messages [Van Renesse 1993]. As an aside, in principle total ordering does not have to imply FIFO ordering, although most group systems that support total ordering also guarantee FIFO ordering. For a more theoretical and in depth treatment of the semantics of group communication see Hadzilacos and Toueg [Hadzilacos and Toueg 1993]

The fourth item in the table, *delivery semantics*, relates to when a message is considered successfully delivered to a group. There are three common choices: *k*-delivery, quorum delivery, and atomic delivery. With *k*-delivery, a broadcast is defined as being successful when *k* processes have received the message for some constant *k*. With quorum delivery, a broadcast is said to be successful when a majority of the current membership has received it. With atomic delivery either all surviving processes receive it or none do. Atomic delivery is the ideal semantics, but is harder to implement since processors can fail.

Item five, *response semantics* deals with what the sending process expects from the receiving processes [Hughes 1989]. There are four broad categories of what the sender can expect: no responses, a single response, many responses, and all responses. Operating systems that integrate

group communication and RPC completely often support all four choices [Cheriton and Zwaenepoel 1985; Birman et al. 1990].

The last design decision specific to group communication is *group structure*. Groups can be either closed or open [Liang et al. 1990]. In a *closed* group, only members can send messages to the group. In an *open* group, nonmembers may also send messages to the group. In addition, groups can be either static or dynamic. In static groups processes cannot leave or join a group, but remain a member of the group for the lifetime of the process. Dynamic groups may have a varying number of members over time; processes can come and go.

If processes can be members of multiple groups, the semantics for *overlapping groups* must be defined. Suppose that two processes are members of both groups G_1 and G_2 and that each group guarantees a total ordering. A design decision has to be made about the ordering between the messages of G_1 and G_2. All choices discussed in this section (none, FIFO, causal, and total ordering) are possible.

To make these design decisions more concrete, we briefly discuss two systems that support group communication. Both systems support open dynamic groups, but differ in their semantics for reliability and ordering. In the V system [Cheriton and Zwaenepoel 1985], groups are identified with a group identifier. If two processes concurrently broadcast two messages, *A* and *B*, respectively, some of the members may receive *A* first and others may receive *B* first. No guarantees about ordering are given. Group communication in the V system is unreliable. Users can, however, build their own group communication primitives with the basic primitives. They could, for example, implement the protocols described in this paper as a library package.

In the Isis system [Birman and Joseph 1987], messages are sent to a group identifier or to a list of addresses. When sending a message, a user specifies how many replies are expected [Birman et al. 1990]. Messages can be totally-ordered, even for groups that overlap. If, for example, processes P_1 and P_2 in Figure 3 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* surviving 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 semantics, but better performance. It is up to the programmer to decide which primitive is required.

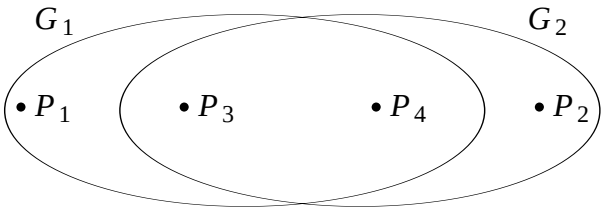


Fig. 3. Total ordering with overlapping groups. P_1 belongs to group G_1. P_2 belongs to group G_2. P_3 and P_4 are member of both groups.

3. Design Choices

Now that we have discussed the design issues in general, let us look at the choices we have made for Amoeba. These are summarized in Figure 4, and will each be discussed below in turn.

Addressing

Groups are addressed by a single address, called a *port*. A port is a large random number. By using a single address per group, a process can send a message to the group without requiring to know which processes (or even how many processes) are members of the group.

Addressing groups with ports fits with Amoeba's client/server model. All services in Amoeba are addressed by ports. When a service is started, it generates a new port and registers the port with the directory service. A client can look up the port using the directory service and asks its own kernel to send a message to the given port. The kernel maps the port onto a network address. If multiple servers listen to the same port, only one (arbitrary) server will get the message. Thus, in Amoeba, processes and groups are addressed in a uniform way.

Issue	**Choice**
Addressing	Group identifier (port)
Reliability	Reliable communication; fault tolerance if specified
Ordering	Total ordering per group
Delivery semantics	All or none
Response semantics	None (RPC is available)
Group structure	Closed and dynamic

Fig. 4. Important design issues of Fig. 3.1 and the choices made in Amoeba.

Reliability

By default, the group primitives provide by default reliable communication, even in the presence of communication failures. If a message is lost due to buffer overflow or hardware error, the protocols will guarantee that the message is resent.

On the user's request, the group primitives can also recover from processor failures. After a processor failure the protocol goes through a recovery phase in which the group is rebuilt from the

processors that are still alive. The protocol guarantees (1) that all the members in the rebuilt group receive all the messages successfully sent by any member of the original group before the failure and (2) that surviving members of the rebuilt group will receive all messages successfully sent by any member of the new group after the failure. If not enough surviving members can be found for rebuilding the group, the recovery phase fails and the group will block until a sufficient number of processors recover. Processors may fail during the recovery algorithm; in this case the recovery algorithm starts again until it succeeds or fails.

To rebuild a group requires consensus on which processors are alive. However, it is known that achieving consensus in an asynchronous distributed system with one faulty processor is impossible [Fischer et al. 1985]. To be able to reach a decision about whether a processor is alive, the algorithm sends messages asking the recipient to respond. If after a certain number of trials a processor does not respond, the processor is declared ‘‘dead’’. Using this method some processors, may be declared dead although they are functioning fine (e.g., when a processor does not respond fast enough). Achieving an approximate consensus is the best one can do in today’s distributed systems.

We decided to make the recovery from processor failures an option, because providing these semantics is expensive and many applications do not need to recover from processor failures. We assume that processors fail due to fail-stop failures [Schneider 1984]. Stronger semantics, such as automatic recovery from Byzantine failures (i.e., processors sending malicious or contradictory messages) and automatic recovery from network partitions, are not supported by the group primitives. Applications requiring these semantics have to implement them explicitly. For a more thorough discussion of the relation between broadcasts semantics and failures, and protocols for different type of failures see Hazdilacos and Toueg [Hadzilacos and Toueg 1993].

Although Amoeba’s network protocol supports unreliable group communication [Kaashoek et al. 1993b], we decided to make only reliable group communication available to the programmer. This has the potential disadvantage that some users pay in performance for semantics that they do not need. It has the advantage, however, that the kernel only has to support one primitive, which simplifies the implementation and makes higher level software more uniform. For the same reason Amoeba also supports only one primitive for point-to-point communication: RPC.

Ordering

The group primitives guarantee a total ordering per group. If two members send messages *A* and *B* concurrently, the protocol guarantees that all members of the group receive either first message *A* and then *B*, or first *B* and then *A*. It never happens that *A* and *B* are interleaved. Many distributed applications are easy to implement with a total ordering, as the programmer can think of processes running in lockstep [Schneider 1990]. Applications using weaker forms of ordering often use a token scheme to guarantee total ordering [Marzullo and Schmuck 1988; Siegel et al. 1990].

We have a simple and efficient protocol for doing reliable totally-ordered group communication. The protocol is presented in detail in the coming sections and is based on an earlier protocol we experimented with [Kaashoek et al. 1989]. There are three key ideas that make our approach feasible. First, to guarantee a total ordering the protocol uses a central machine per group, called the *sequencer*.

If the sequencer crashes, the remaining group members elect a new one. Second, the protocol is based on a *negative acknowledgement* scheme. In a negative acknowledgement scheme, a process does not send an acknowledgement as soon as it receives a message. Instead, it sends a negative acknowledgement as soon as it discovers that it has missed a message. Third, acknowledgements are piggybacked on regular data messages to further reduce the number of protocol messages. These ideas are well known techniques. Chang and Maxemchuck, for example, discuss a protocol similar to ours that also combines these three ideas [Chang and Maxemchuk 1984].

Although at first sight it may seem strange to use a *centralized* sequencer in a *distributed system*, this decision is attractive. First, distributed protocols for total ordering are in general more complex and perform less well. Second, today's computers are very reliable and it is therefore unlikely that the sequencer will crash. The major disadvantage of having a sequencer is that the protocol does not scale to very large groups. In practice, however, this drawback is minor. The sequencer totally orders messages for a single group, not for the whole system. Furthermore, the sequencer performs a simple and computationally unintensive task and can therefore process many hundreds of messages per second. For many applications hundreds of messages per second is sufficient.

There are two reasons for using a negative acknowledgement scheme. First, in a *positive acknowledgement scheme*, a process sends an acknowledgement back to the sender as soon as it receives the message. This works fine for point-to-point messages, but not for broadcast messages. If in a group of 256 processes, a process sends a broadcast message to the group, all 255 acknowledgements will be received by the sender at approximately the same time. As network interfaces can only buffer a fixed number of messages, a number of the acknowledgements will be lost, leading to unnecessary timeouts and retransmissions of the original message. Second, today's networks are very reliable and network packets are delivered with a very high probability. Thus not sending acknowledgements at all, but piggybacking them on regular data messages is feasible. Another alternative would be to use a positive acknowledgement scheme, but force the receivers to wait some ''random'' time before sending an acknowledgement [Danzig 1989]. This approach is attractive in unreliable networks, but it causes far more acknowledgements to be sent than with a negative acknowledgement scheme.

Delivery Semantics and Response Semantics

Per default, the group communication primitives deliver a message to all destinations, even in the face of communication failures. On the user's request, the primitives can also guarantee ''all-or-none'' delivery in the face of processor failures. The protocols for providing these semantics are more expensive, and hence we decided to make it an option. Users can trade performance for fault tolerance.

When a member receives a broadcast message, there is no group primitive available to send a reply. For the request/response type of communication, RPC is available.

Group Structure

Unlike many other systems, we have chosen to use closed groups. A process that is not a member and that wishes to communicate with a group can perform an RPC with one of the members (or it can join the group). One reason for doing so is that a client 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 whether the client consists of a single process or a group of processes. This design decision is in the spirit of the client-server paradigm: a client knows what operations are allowed, but should not know how these operations are implemented by the service; the client should not have to know whether it should use RPC or group communication to communicate with a service.

A second reason for closed groups is that it makes an efficient implementation of totally-ordered reliable broadcast possible. To implement the protocol, state is maintained for each member. If all processes can send messages to a group, they all have to keep state information about the groups that they are communicating with. Furthermore, the members also have to keep state for all the processes that are communicating with the groups. To make it possible to control the amount of state needed to implement the protocol, we decided on closed groups.

In Isis, this problem is solved in a different way. Isis presents the user with open groups, but implements it using closed groups. When a process wants to communicate with a group, the system either performs a join or an RPC with one of the members. In the latter case, the member broadcasts the message to the whole group. Thus, although the user has the illusion of open groups, the current implementation of Isis uses only closed groups.

A third reason for closed groups is that they are as useful as open groups. Just as in Isis, one can simulate an open group in Amoeba. A process performs an RPC with one of the members of the group. If a member receives an RPC, it broadcasts the request to the rest of the group. Compared to real open groups, the cost is that a request goes twice over the network instead of once.

Figure 5 shows a very small Amoeba system, with 12 processes and 3 groups, and how they interact. The parallel application replicates shared data using group communication to reduce access time. If the application wants to store an object with the directory service, it uses RPC to communicate with it. One of the directory servers will get the request. The directory server uses group communication to achieve fault tolerance and high availability. To store results on disk, a directory server communicates with the disk service, using RPC. The disk service may again use group communication internally for availability and fault tolerance (currently not done). Each application or service may be built out of one process or a group of processes which communicate with other services using RPC.

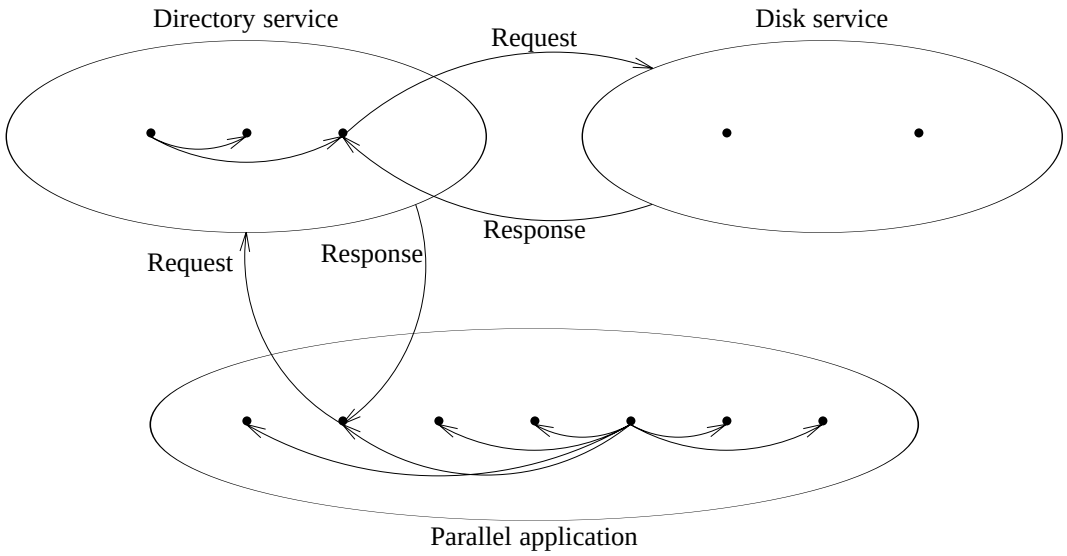


Fig. 5. An example Amoeba system with processes (dots), groups (ellipses), and their interaction (arrows). A request/response pair makes up one RPC. The request is sent to a port identifying a service and one of the servers will take the request; the corresponding response is sent back to the process doing the RPC.

4. Group Primitives in Amoeba

The primitives to manage groups and to communicate within a group are listed in Figure 6. We will now discuss each primitive in turn.

A group is created by calling *CreateGroup*. The creator of the group is automatically a member of the group. The first parameter is a port identifying the group. The second parameter is the number of member-crashes the group must be able to 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 simplifies the implementation: the maximum number of members, the number of buffers the protocol can use, and the maximum message size. Using this information, the kernel allocates memory for buffering messages and for member information. (Although these parameters could easily be replaced by default values, we decided against this for the sake of flexibility.) If not enough memory is available, *CreateGroup* fails. Otherwise, it succeeds and returns a small integer, called a group descriptor, *gd*, which is used to identify the group in subsequent group calls.

Once a group with port *p* has been created, other processes can become members of it by calling *JoinGroup* with the port *p*. (The port is part of the Amoeba header *hdr*.) Only processes that know port *p* can join the group. When a message is sent to a group, only the group members receive the message. Like *CreateGroup*, *JoinGroup* returns a group descriptor for use in subsequent group calls. In addition to adding a process to a group, *JoinGroup* delivers a small message, *hdr*, to all other members. In this way, the other members are told that a new member has joined the group.

Function(parameters) → result	**Description**
CreateGroup(port, resilience, max_group, nr_buf, max_msg) → gd	Create a group. A process specifies how many member failures must be tolerated without loss of any message.
JoinGroup(hdr) → gd	Join a specified group.
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 totally 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 newly reset group has at least *nr_member* members, it succeeds.
GetInfoGroup(gd, &state)	Return state information about the group, such as the number of group members and the caller's member id.
ForwardRequest(gd, member_id)	Forward a request for the group to another group member.

Fig. 6. Primitives to manage a group and to communicate within a group. A message consists of a header (a small message) and a buffer (a linear array of bytes). The header contains the port of a group. An output parameter is marked with "&".

Once a process is a member of a group, it can leave the group by calling *LeaveGroup*. Once a member has left the group, it does not receive subsequent broadcasts. In addition to causing the process to leave the group, *LeaveGroup* delivers *hdr* to all other members. In this way, the other members are told that a member has left. The member receives its own message, so that it can check whether it has processed all messages up to its leave message. The last member calling *LeaveGroup* automatically causes the group to vanish.

To broadcast a message, a process calls *SendToGroup*. This primitive 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 *r*, the protocol guarantees that even in the event of simultaneous crashes of up to *r* members, it will either deliver the message to all remaining members or to none. Choosing a large value for *r* provides a high degree of fault tolerance, but this extracts a cost in terms of performance. *SendToGroup* blocks until $r + 1$ members have received the message. The tradeoff 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 receive *A* first and then *B*, or
2. All members receive *B* first and then *A*.

Random mixtures, where some members get *A* first and others get *B* first are guaranteed not to occur.

To receive a broadcast, a member calls *ReceiveFromGroup*, which blocks it until the next message in the total order arrives. 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 (no more messages can be sent to the group), because it may run out of buffers. Messages are never permanently discarded until received by all members.

ResetGroup allows recovery from member crashes. If one of the members (or its kernel) is unreachable, it is deemed to have crashed and the protocol enters a recovery mode. In this mode, it only accepts messages needed to run the recovery protocol and all outstanding *ReceiveFromGroup* calls return an error value that indicates a member crash. Any member can now call *ResetGroup* to transform the group into a new group that contains as many surviving members 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* to all new members. It may happen that multiple members initiate a recovery at the same moment. The new group is built only once, however, and consists of all the members that can communicate with each other. The *hdr* is also delivered only 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 the group is alive. In our system, the user is able to decide on the policy. The group primitives provide only the mechanism.

GetInfoGroup allows a group member to obtain information about the group from its kernel. The call returns information such as the number of members in the group and the caller's member id. Each group member has a unique member id.

The final primitive, *ForwardRequest*, integrates RPC with group communication. When a client does an RPC to a service, the client has no idea which server will get the request; it goes to one of the servers, effectively at random. If the server that gets the request is not able to handle the request (e.g., because it does not have the data requested), it can forward the request to another server in the group (*member_id* specifies the server). The forwarding occurs transparently to the client. The client cannot even tell that the service is provided by multiple servers.

To summarize, the group primitives provide an abstraction that enables programmers to design applications consisting of one or more processes running on different machines. It is a simple, but powerful, abstraction. All members of a single group see all events concerning this group 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 totally-ordered. If, for example, one process calls *JoinGroup* and a member calls *SendToGroup*, either all members first receive the join and then the broadcast or all members 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 orderings 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.

5. The Broadcast Protocol

The protocol to be described runs inside the kernel and is accessible through the primitives described in the previous section. It assumes that *unreliable* message passing between entities is possible; fragmentation, reassembly, and routing of messages are done at lower layers in the kernel [Kaashoek et al. 1993b]. The protocol performs best on a network that supports hardware multicast. Lower layers, however, treat multicast as an optimization of sending point-to-point messages. If multicast is not available, then point-to-point communication will be used. Even if only point-to-point communication is available, the protocol is in most cases still more efficient than performing n RPCs. In a mesh interconnection network, for example, the routing protocol will ensure that the delay of sending n messages is only on the order of $\log_2 n$.

Each kernel running a group member maintains 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 group member can, at any instant, decide to broadcast a message to its group. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communication, lost packets, finite buffers, and node failures.

Without loss of generality, we assume for the rest of this section that the system contains one group, with each member running on a separate processor (see Fig. 7). When the application starts up, 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 (this procedure is described in Section 5.3). The sequencer machine is in no way special—it has the same hardware and runs the same kernel as all the other machines. The only difference is that it is currently performing the sequencer function in addition to its normal tasks.

5.1. Basic Protocol

A brief description of the basic protocol is as follows (a complete description is given in the next section). When a group member calls *SendToGroup* to send a message, M, it hands the message to its kernel and blocks. The kernel encapsulates M in an ordinary point-to-point message and sends it to the sequencer. When the sequencer receives M, it allocates the next sequence number, s, and

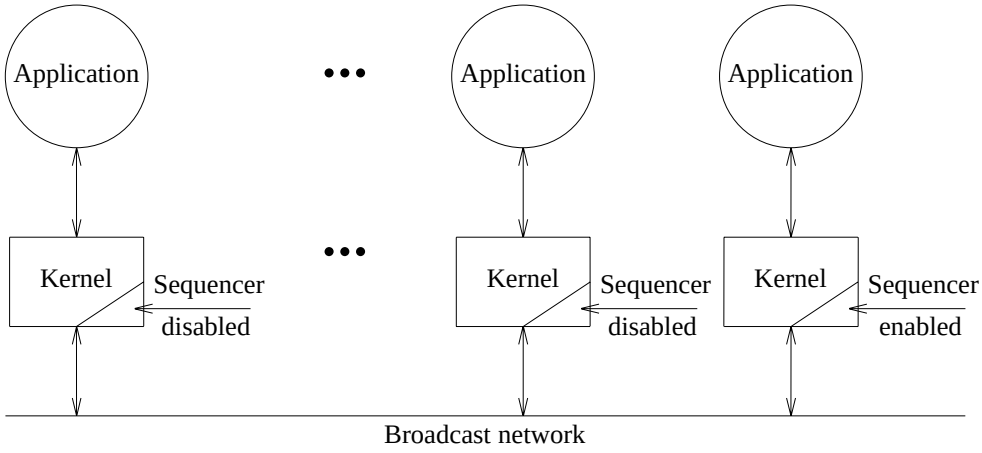


Fig. 7. System structure. Each node runs a kernel and a user application. Each kernel is capable of being sequencer, but, at any instant, only one of them functions as sequencer. If the sequencer crashes, the remaining nodes can elect a new one.

broadcasts a message containing M and s. Thus all broadcasts are issued from the same node, the sequencer. Assuming that no messages are lost, it is easy to see that if two members concurrently want to broadcast, one of them will reach the sequencer first and its message will be broadcast first. Only when that broadcast has been completed will the other broadcast be started. Thus, the sequencer provides a total ordering.

When the kernel that sent M 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 message eventually arrives, the kernel will immediately notice a gap in the sequence numbers. If it was expecting s next, and it receives $s + 1$ instead, it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for a copy 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 kernels also keep a history buffer, to be able to recover from sequencer failures and to buffer messages when there is no outstanding *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 indefinitely. However, if it could somehow discover that all members have received broadcasts up to and including m, it could then purge the broadcast messages up to m 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 se-

quence number of the last broadcast received by the sender of the message (i.e., a piggybacked acknowledgement). 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 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, the 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 send the sequencer a special message acknowledging all received broadcasts. The sequencer can also request this information when it runs out of space in its history buffer.

PB Method and BB Method

There is a subtle design point in the protocol; there are actually two ways to do a broadcast. In the method we have just described, the sender sends a point-to-point message to the sequencer, which then broadcasts it. We call this the *PB method* (Point-to-point followed by a Broadcast). In the *BB method*, 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 the PB method, each message appears on the network twice: once to the sequencer and once from the sequencer. Thus a message of length *n* bytes consumes 2*n* bytes of network bandwidth. However, only the second message is broadcast, so each user machine is interrupted only once (for the second message).

In the BB method, 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 the PB method wastes bandwidth to reduce the number of interrupts and the BB method minimizes bandwidth usage at the cost of more interrupts. For messages up to *n* bytes, where *n* is chosen by the user, the protocol uses the PB method and for messages larger than *n* bytes the protocol uses the BB method.

Processor Failures

The protocol described so far recovers from communication failures, but does not guarantee that all surviving members receive all messages that were sent before a member crashed. For example, suppose a member sends a message to the sequencer, which then broadcasts it. The sender receives the broadcast and delivers it to the application, which interacts with the external world. Now assume all other processes miss the broadcast, and the sender and sequencer both crash. The effects of the message are visible but none of the other members will ever receive it. This is a dangerous situation that can lead to all kinds of disasters because the ‘‘all-or-none’’ semantics have been violated.

To avoid this situation, *CreateGroup* has a parameter, *r*, the *resilience degree*, which specifies the resiliency. This means that the *SendToGroup* primitive does not return control to the application until the kernel knows that at least *r* other kernels have received the message. To achieve this, a kernel sends the message to the sequencer point-to-point (PB method) or broadcasts the message to the group (BB method). The sequencer allocates the next sequence number, but does not officially accept the message yet. Instead, it buffers the message and broadcasts the message and sequence number as a request for broadcasting to the group. When a member kernel receives this message, it buffers the message in its history and if its member identifier is lower than *r*, it sends an acknowledgement message to the sequencer. Any *r* members besides the sending kernel would be fine, but to simplify the implementation we pick the *r* lowest-numbered. After receiving these acknowledgements, the sequencer broadcasts the *accept* message. Only after receiving the *accept* message can members other than the sequencer deliver the message to the application. That way, no matter which *r* machines crash, there will be at least one surviving member containing the full history, so everyone else can be brought up-to-date during the recovery. Thus, an increase in fault tolerance is paid for by a decrease in performance. The tradeoff chosen is up to the user.

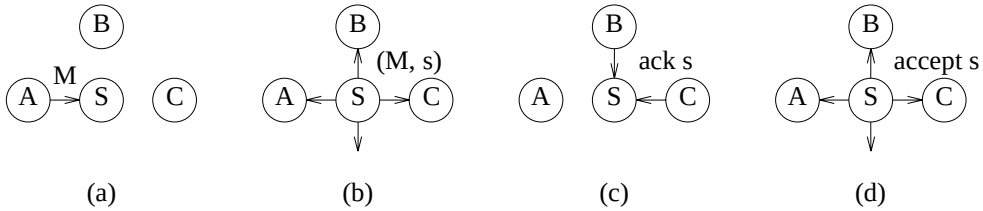


Fig. 8. PB protocol for $r = 2$.

The PB and BB method for $r = 2$ are illustrated in Figure 8 and in Figure 9 respectively. In Figure 8(a), machine A sends a message, *M*, to the sequencer, where it is assigned sequence number *s*. The message (containing the sequence number *s*) is now broadcast to all members and buffered (Fig. 8(b)). The *r* lowest-numbered kernels (say, machines *B* and *C* in Figure 8(c)) send an acknowledgement back to the sequencer to confirm that they have received and buffered the message with sequence number *s*. After receiving the *r* acknowledgements, the message with sequence number *s* is officially accepted and the sequencer broadcasts a short accept message with sequence number *s* (Fig. 8(d)). When a machine receives the accept message, it can deliver the message to the application.

The BB method for $r = 2$ is very similar to the PB method (see Fig. 9); only the events in (a) and (b) differ. Instead of sending a point-to-point message to the sequencer, machine *A* broadcasts the message to the whole group (Fig. 9(a)). When the sequencer receives this message, it allocates the next sequence number and broadcasts it (Fig. 9(b)). From then on the BB method is identical to the PB method. Thus, the important difference between the PB and BB method is that in the PB method

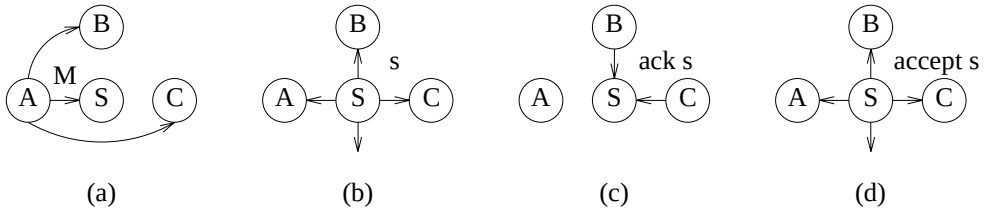


Fig. 9. BB method for $r = 2$.

the message *M* goes over the network twice, while in the BB method it goes only over the network once.

At first sight, it may seem that a more efficient protocol can be used for $r > 0$. Namely, a kernel broadcasts the message to the group. On receiving a broadcast, *r* lowest-numbered kernels immediately buffer the message in their history and send acknowledgement messages to the sequencer, instead of waiting until the sequencer announces a sequence number for the broadcast request. After receiving the acknowledgements, the sequencer broadcasts the *accept* message. This protocol would save one broadcast message (the message from the sequencer announcing the sequence number for the broadcast request).

This protocol is, however, incorrect. Assume that *r* kernels have buffered a number of messages and have sent an acknowledgement for each of them and that all *accept* messages from the sequencer are lost. The following could now happen. The sequencer delivers the message to its application, and then the sequencer (and application) crashes. During recovery, the remaining members would have no way of deciding how the buffered messages should be ordered, violating the rule that all messages should be delivered in the same order. Even if they decide among themselves on some order, they could potentially deliver the messages in a different order than the sequencer did and still violate the rule. To avoid this situation, the sequencer announces the sequence number for a message before *r* kernels send an acknowledgement.

It is interesting to see how Isis deals with this situation. In Isis it may happen that after a processor failure, messages on the remaining processors are delivered in a different order than was done on the failed processor. If an application requires stronger semantics, it is up to the programmer to call a primitive that blocks the application until it is known that all other kernels will deliver the message in the same order [Birman et al. 1990].

In summary, there are two methods of sending a reliable totally-ordered broadcast, PB and BB. The PB method and the BB method are logically equivalent but have different performance characteristics. (In Section 6.2 we will give a detailed comparison between the PB method and the BB method.) For $r > 0$, additional messages are needed to guarantee that broadcasts are delivered in the same order, even in the face of processor failures.

5.2. Protocol during Normal Operation

In this section, we will describe in detail how the sender, sequencer, and receivers behave during normal operation (no member failures).

Data Structures

Figure 10 shows the data structures used by the protocol. Each kernel keeps state information for each of its members. The information stored for each member consists of general information, membership information, and history information. The general information includes the port of the group to which the member belongs, the network address for the group, on what message size to switch from the PB method to the BB method, the current state of the protocol (e.g., receiving, sending, etc.), the resilience degree (*r*), and the current incarnation number of the group. The resilience degree, *g_resilience*, specifies how many concurrent failures the group must tolerate without losing any messages. It is specified when the group is created. The incarnation number of a group, *g_incarnation*, is incremented after recovery from a member failure. Each message sent is stamped with the current incarnation number and is only processed if it is equal to *g_incarnation*; otherwise it is discarded. If no member failures happen, then *g_incarnation* stays at 0.

The membership information consists of the list of members, the total number of members, the identity of the current sequencer, and the identity of the current coordinator (only used during recovery from member failures). Furthermore, the kernel stores the member identifier, *g_index*, for this member and its rank, *g_memrank*. The member id does not change during the time the application is member of a group. The rank is used to decide if a kernel should send an acknowledgement back when a broadcast request arrives and the *resilience degree* is higher than 0. The rank of a member can change during its lifetime. If, for example, a group consists of three members, numbered 0, 1, and 2 respectively, the ranks for these members are initially equal to the member ids. If now, for example, member 1 leaves, then the rank of member 2 changes to 1. In this way, it is easy for each member to decide whether it belongs to the *r* lowest members. Since every member is guaranteed to receive all join and leave events in the same order, this information will always be consistent.

Each kernel maintains an array of structs (*member*). The array is indexed by member identifier, *m*. The struct contains the sequence number expected by *m*, *m_expect*, the last message number used by *m*, *m_messid*, and *m*'s network address. The sequencer uses the *m_expect* fields to determine which messages can be safely removed from the history buffer. The message number, *m_messid*, gives the last message number received from a member and is used to detect duplicates generated by timeouts. The *retry counter*, *m_retrial*, is used to determine when a member has failed. If a kernel is waiting for a reply from another kernel and it does not receive the message within a certain time frame, the counter is decremented. If it reaches zero, the kernel is considered to be down. The other fields are used only during recovery (see Section 5.3).

The history information consists of a circular buffer with a number of indices telling how big the buffer is and which part of the buffer is filled. Each buffer in the history contains a complete mes-

```
/* On each machine, a struct for each group is maintained. */
struct group {                                  /* general info */
   port g_port;                                 /* a port identifies a group */
   adr_t g_addr;                                /* group network address */
   int g_large;                                 /* dividing line between PB and BB */
   long g_flags;                                /* protocol state: FL_RECOVER, … */
   int g_resilience;                            /* resilience degree */
   short g_incarnation;                         /* incarnation number */

   /* Member information */
   int g_total;                                 /* group size */
   struct member *g_member;                     /* list of members */
   struct member *g_me;                         /* pointer to my member struct */
   struct member *g_seq;                        /* pointer to sequencer struct */
   struct member *g_coord;                      /* pointer to coordinator struct */
   int g_index;                                 /* my index */
   int g_memrank;                               /* member rank */

   /* History information in circular buffer */
   hist_p g_history;                            /* history of bcast messages */
   int g_nhist;                                 /* size of history */
   int g_nextseqno;                             /* next sequence number */
   int g_minhistory;                            /* lowest entry used */
   int g_nexthistory;                           /* next entry to store message */
};

/* On each machine, a struct for each member is maintained */
struct member {
   adr_t m_member;                              /* member network address */
   int m_expect;                                /* sequence number expected by member */
   int m_messid;                                /* next message id to use */
   int m_retrial;                               /* retry counter */
   int m_vote;                                  /* vote for this member (recovery) */
   int m_replied;                               /* has the member replied? */
};

/* Broadcast protocol header */
struct bc_hdr {
   short b_type;                                /* type: BC_BCASTREQ, … */
   short b_incarnation;                         /* incarnation number */
   int b_seqno;                                 /* global sequence number */
   int b_messnr;                                /* message identifier */
   int b_expect;                                /* sequence number expected by application */
   int b_cpu;                                   /* member identifier */
};
```

Fig. 10. Declarations used by the protocol.

sage including the user-supplied data (the upper bound on the size of a message is passed as an argument when the group is created). The history buffer consists of three parts (see Fig. 11). The circular part between *g_nexthistory* and *g_minhistory* consists of free buffers. The circular part between

g_minhistory and *g_nextseqno* contains messages that have been sequenced, but are buffered to be delivered to the user application or so that the kernel can respond to retransmission requests. The circular part between *g_nextseqno* and *g_nexthistory* is used by the sequencer to buffer messages for which it does not know yet if all members have room to store them in their history buffers. After synchronizing with the other history buffers, the sequencer will take the buffered requests and process them (broadcast an official accept). The ordinary kernels use the third part to buffer messages that are received out-of-sequence until the missing messages are received. Buffering messages in the third part of the history buffer avoids unnecessary retransmission of those messages later on.

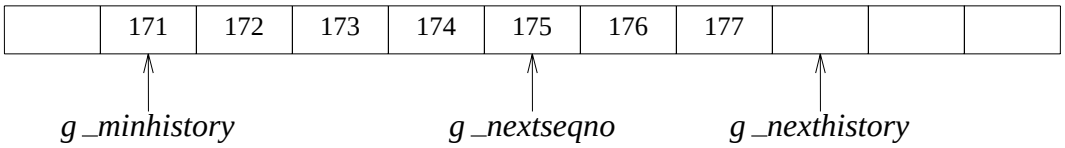


Fig. 11. The history buffer has three parts: (1) free buffers; (2) messages that have been sequenced but are buffered until they can be delivered to the user or are buffered for retransmissions; (3) messages that are buffered because the sequencer does not know if the other kernels have room in their history to store the message. Ordinary members use the third part to buffer messages that arrive out-of-sequence.

Each message sent by the protocol contains a fixed size protocol header, consisting of six fields. The *b_type* field indicates the kind of a message (see Fig. 12). The *b_incarnation* gives the incarnation of the member that is sending the message. The *b_seqno* field is used by the sequencer to sequence broadcasts. The *b_messnr* and *b_cpu* together uniquely identify a message. They are used to detect duplicates. The *b_expect* field is used to piggyback the acknowledgement for the last broadcast delivered so far. When receiving a message, a kernel updates *m_expect* with *b_expect*. If the sequencer knows that *m_expect* for each member is larger than *g_minhistory*, it can increase *g_minhistory* and thereby free history buffers.

Receiving a Message

Let us now look at the protocols for receiving and sending messages reliably. When a member wants to receive a broadcast, it invokes its local kernel by calling *ReceiveFromGroup* and passes it pointers to a header and a buffer (see Fig. 13). (For simplicity we left out the code that checks the parameters as well as the code dealing with the transition from user space to kernel space, and back.) The kernel checks the history to see if there are any messages buffered that can be delivered. If there are none, the thread calling *ReceiveFromGroup* is blocked until a message can be delivered. If a deliverable message is present, it is copied from the history buffer into the buffer supplied by the application. The number of bytes actually received is returned to the application process.

Each time a broadcast comes in from the sequencer, the kernel checks to see if there is a thread waiting to receive a message. If so, it unblocks the thread and gives it the message.

Type	From	To	Description
BC_JOINREQ	Member	Sequencer	Request to join the group
BC_JOIN	Sequencer	Group	Accept join message
BC_LEAVEREQ	Member	Sequencer	Request to leave the group
BC_LEAVE	Sequencer	Group	Accept leave message
BC_BCASTREQ	Member	Sequencer or group	Request to broadcast
BC_BCAST	Sequencer	Group	Accept broadcast message
BC_ACK	Member	Sequencer	Message is received (if r > 0)
BC_RETRANS	Member	Sequencer	Request asking for missed message
BC_SYNC	Sequencer	Group	Synchronize histories
BC_STATE	Member	Sequencer	Tell next expected sequence number
BC_ALIVEREQ	Member	Member	Check if destination is alive
BC_ALIVE	Member	Member	Acknowledgement of BC_ALIVEREQ
BC_REFORMREQ	Coordinator	Group	Request for entering recovery mode
BC_VOTE	Member	Coordinator	Vote for new sequencer
BC_RESULT	Coordinator	Group	Result of the voting phase
BC_RESULTACK	Member	Coordinator	Ack for receiving the final vote

Fig. 12. Possible values for *b_type* and their function.

Sending a Message

When a member wants to do a broadcast, it invokes its local kernel by calling *SendToGroup*, passing as parameters a header and data. The kernel then executes the algorithm given in Figure 14. It builds a message consisting of the protocol header, and the user-supplied header and data. The kernel sets *b_type* to BC_BCASTREQ, *b_cpu* to the member's id, *b_incarno* to the current incarnation, and *b_expect* to the value of the member's *m_expect*. Depending on the size of the data and *g_large*, the kernel sends the message point-to-point to the sequencer (PB method) or broadcasts the message to the group (BB method). The default value for *g_large* is equal to the maximum size of a network packet. Thus, small messages that fit in one network packet are sent using the PB method and larger messages are sent using the BB method. The programmer may override this default value. Once the message is sent, the kernel blocks the sending member until the message comes back from the

```
long ReceiveFromGroup(gd, hdr, buf, cnt, more)
   int gd;                                          /* group descriptor */
   struct header *hdr;                              /* pointer to Amoeba header buffer */
   char *buf;                                       /* pointer to empty data buffer */
   long cnt;                                        /* size of data buffer */
   int *more;                                       /* pointer to more flag */
{
   struct group *g;                                 /* pointer to group structure */
   struct hist *h;                                  /* pointer into the history */
   long rs;                                         /* size of the message to be received */

   g = groupindex[gd];                              /* set group pointer */
   g–>g_flags |= FL_RECEIVING;                      /* start receiving */
   while(!HST_IN(g, g–>g_me–>m_expect)) {           /* is there a buffered message? */
        if (g–>g_flags & FL_RESET) {                /* don't block during recovery */
                g–>g_flags &= ~FL_RECEIVING;        /* switch flag off */
                return(BC_ABORT);                   /* return failure */
        }
        block(g);                                   /* no, wait until one comes in */
   }
   g–>g_flags &= ~FL_RECEIVING;                     /* switch flag off */
   h = &g–>g_history[HST_MOD(g, g–>g_me–>m_expect)];/* get it */
   bcopy(h–>h_data, hdr, sizeof(struct header));    /* copy to user space */
   rs = MIN(cnt, h–>h_size – sizeof(struct header)); /* MIN(a,b) = (a < b ? a : b) */
   bcopy(h–>h_data + sizeof(struct header), buf, rs); /* copy to buf */
   g–>g_me–>m_expect++;                             /* message is now delivered */
   *more = g–>g_nextseqno – g–>g_me–>m_expect;      /* number of buffered messages */
   return(rs);                                      /* return number of bytes received */
}
```

Fig. 13. Algorithm used by kernel to receive a reliable broadcast.

sequencer or until it receives a timeout. If, after *n* retries, no message is received from the sequencer, the member assumes that the sequencer has crashed and enters recovery mode (see Section 5.3). During recovery it is determined if the send failed or succeeded.

Protocol

Having looked at what the sender does to transmit a message to the sequencer for broadcast, let us now consider what a kernel does when the BC_BCASTREQ message comes in (see Fig. 15). If the message is sent using the BB method, then all members will receive the broadcast request (absent a network failure). If $b\_seqno = 0$, the broadcast request is sent using the BB method and the ordinary members will buffer the request until the sequencer broadcasts an accept message. If $b\_seqno > 0$, the sequencer has sequenced the message, but did not accept the message yet, because it is waiting for *r* acknowledgements. In this case, the members store the message in the third part of the history and send an acknowledgement (BC_ACK) if their rank is less than or equal to *r*. This informs the sequencer that the message sent by *b_cpu* with message number *b_messnr* has been received. Once the

```
int SendToGroup(gd, hdr, data, size)
   int gd;                                          /* group descriptor */
   struct header *hdr;                              /* pointer to header to be sent */
   char *data;                                      /* pointer to data to be sent */
   long size;                                       /* size of data */
{
   struct group *g;                                 /* pointer to group structure */
   long messid;                                     /* messid for the message to be sent */
   struct pkt *msg;                                 /* the message to be sent */

   g = groupindex[gd];                              /* set pointer */
   if (g–>g_flags & FL_RESET) return(BC_ABORT);/* don't send during recovery */

   g–>g_seq–>m_retrial = g–>g_maxretrial;           /* set maximum number of retries */
   messid = g–>g_me–>m_messid+1;                    /* set message identifier */
   g–>g_flags |= FL_SENDING;                        /* start sending message */
   do {
        setmsg(&msg, BC_BCASTREQ, −1, messid, g–>g_index, g–>g_me–>m_expect,
             g–>g_incarnation, hdr, (long) data, size);  /* build message */
        set_timer(g, msg, settimer);                /* set timer */
        if (g–>g_seq == g–>g_me) sendlocal(g, msg);/* am I the sequencer? */
        else if (size >= g–>g_large) multicast(&g–>g_addr, msg);/* use BB method? */
        else unicast(&g–>g_seq–>m_addr, msg);       /* use PB method */

        /* Block until broadcast succeeds, fails, or times out. */
        block(g);                                   /* suspend calling thread */
        if (g–>g_flags & FL_SENDING) {              /* timeout? */
                g–>g_seq–>m_retrial−−;              /* decrease retry counter */
                if (g–>g_seq–>m_retrial <= 0) recover(g);/* did the sequencer crash? */
        }
   } while(g–>g_flags & FL_SENDING);                /* done? */
   return(g–>g_me–>messid ≥ messid ? BC_OK : BC_FAIL);/* return success or failure */
}
```

Fig. 14. Algorithm used by sending kernel to achieve reliable broadcast.

sequencer has received *r* of these acknowledgements, it will accept the message officially and broadcast a short accept message (BC_BCAST without data).

When the sequencer receives the broadcast request, it updates its table with member information and it tries to free some buffers in its history using the piggybacked acknowledgements. Then, it checks if the message is a duplicate by examining *b_cpu* and *b_messnr*. If so, it informs the sender that the message already has been sent.

If the message is new and $r = 0$, the sequencer stores the message in its history and officially accepts the message (i.e., the sequencer changes the type of the message from BC_BCASTREQ to BC_BCAST). If $r > 0$, the sequencer stores the message in its history buffer, but it does not accept the message officially. Instead, it forwards the request with sequence number to the group and waits for *r* acknowledgements.

The history is processed each time a broadcast request is received, *r* acknowledgements for a

```
void bcastreq(g, bc, data, n)
   group_p g;                                          /* pointer to group structure */
   struct bc_hdr *bc;                                  /* pointer to protocol header */
   char *data;                                         /* pointer do data in message */
   int n;                                              /* number of bytes in data */
{
   struct hist *hist;                                  /* pointer in history */
   struct member *src = g–>g_member + bc–>b_cpu;/* pointer to the sender */
   struct pkt *msg;                                    /* reply message */

   if (g–>g_me != g–>g_seq) {                          /* am I the sequencer? If no: */
        if(bc–>b_seqno == 0) {
                /* A request without seqno has been received. This must be the BB method. */
                mem_buffer(g, src, bc, data, n);       /* buffer it */
        } else {
                /* A request with seqno; this must be for resilience > 0. Store the message
                 * in the right place in the history and send an ack if my rank <= resilience. */
                store_in_history_and_send_ack(g, bc, data, n);
        }
        return;                                        /* done */
   }

   /* Yes, the sequencer. This must be the PB protocol. */
   if (src–>m_expect < bc–>b_expect)                   /* update member info? */
        src–>m_expect = bc–>b_expect;
   if (hst_free(g)) g–>g_flags &= ~FL_SYNC;            /* is synchronization needed? */
   if (bc–>b_messnr <= src–>m_messid) {                /* old request? */
        /* Send sequence number back as an ack to the sender. */
        retrial(g, bc–>b_cpu);
   } else if (HST_FULL(g)) {                           /* history full? */
        g–>g_flags |= FL_SYNC;                         /* synchronize */
        synchronize(g);                                /* multicast a BC_SYNC messages */
   } else {                                            /* append message to history */
        src–>m_messid = bc–>b_messnr;                  /* remember messid */
        bc–>b_seqno = g–>g_nexthistory;                /* assign sequence number */
        if (g–>g_resilience == 0) bc–>b_type = BC_BCAST;/* accept request */
        /* Append to history and increase g_nexthistory. */
        hist = hst_append(g, bc, data, n);
        if (g–>g_resilience > 0) {                     /* resilience degree > 0? */
                forward_msg_to_members(g, hist);       /* forward request with seqno */
        } else hist–>h_accept = 1;                     /* message is accepted */
        processhist(g);                                /* accept the new broadcast */
   }
}
```

Fig. 15. Algorithm executed by all kernels (including sequencer) when a BC_BCASTREQ message arrives.

buffered broadcast message have been received, or when the sequencer learns from a piggybacked acknowledgement that it can free buffers to accept a buffered request (a message stored in the third

part of the history). *Processhist* accepts broadcast messages buffered in the history as long as the history does not fill up (see Fig. 16). It takes the next unprocessed message from the history buffer, broadcasts the message (PB method) or broadcasts an accept message (BB method) and increases *g_nextseqno*. At this point the message has officially been accepted.

If the message just accepted was sent by a member running on the sequencer's kernel, the member can be unblocked and the *SendToGroup* returns successfully.

```
void processhist(g)
   struct group *g;                                  /* pointer to group structure */
{
   struct hist *hist;                                /* pointer into the history */
   struct member *src;                               /* pointer to the sender */
   struct *msg;                                      /* reply message */

   for(hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)];/* get first msg */
      g–>g_nextseqno < g–>g_nexthistory && hist->h_accept;/* process msg? */
      hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)]) {/* get next msg */
        if (!HST_SYNCHRONIZE(g)) {                   /* synchronize first? */
              src = &g–>g_member[hist–>h_bc.b_cpu];  /* set the sender */
              g–>g_nextseqno++;                      /* accept broadcast officially */
              if (src != g–>g_seq && hist–>h_size >= g–>g_large) {/* BB method? */
                    /* Build accept message and multicast. */
                    buildmsg(&msg, 0, &hist–>h_bc, 0, 0);
                    multicast(msg, &g–>g_addr);
              } else {                               /* PB method */
                    /* Build complete message and multicast it. */
                    buildmsg(&msg, 0, &hist–>h_bc, hist–>h_data,  hist–>h_size);
                    multicast(msg, &g–>g_addr);
              }
              if ((g–>g_flags & FL_SENDING) && hist–>h_bc.b_cpu == g–>g_index) {
                    /* Message was sent by a member running on the sequencer kernel. */
                    g–>g_flags &= ~FL_SENDING;     /* switch flag off */
                    unblock(g);                    /* unblock application */
              }
        } else  {
              g–>g_flags |= FL_SYNC;               /* synchronize */
              synchronize(g);                      /* multicast BC_SYNC message */
              break;                               /* stop processing */
        }
   }
}
```

Fig. 16. Algorithm executed by the sequencer to process the history.

The protocol requires three algorithms to be executed. First, the sender must build a message and transmit it to the sequencer (PB) or broadcast it (BB). Second, the sequencer and members must process incoming BC_BCASTREQ messages. The sequencer broadcasts the messages with sequence number (PB) or broadcasts a short accept message (BB); the members buffer them until an official accept from the sequencer arrives (BB). Third and last, the members must handle arriving BC_BCAST

messages from the sequencer. We have already described the first two steps. Now let us look at the last one.

```
void broadcastreceive(g, bc, data, n)
   struct group *g;                                  /* pointer to group state */
   struct bc_hdr *bc;                                /* pointer to protocol header */
   char *data;                                       /* pointer to data */
   long n;                                           /* number of bytes in data */
{
   int received = 1;                                 /* are data received? */
   struct member *src;                               /* pointer to original sender */
   struct hist *hist;                                /* pointer into the history */

   src = g–>g_member + bc–>b_cpu;                    /* set source */
   /* If the PB method is used, the message contains the original data; otherwise
    * the message is only a short accept msg and the data should have been received
    * and is buffered. */
   if (n == 0) {                                     /* short accept msg? */
        /* Yes, the BB method or resilience degree > 0. */
        hist = &g–>g_history[HST_MOD(g, bc–>b_seqno)];
        if (g–>g_resilience > 0 && hist–>h_bc.m_messid == bc–>b_messnr) {
                /* The message is stored as BC_BCASTREQ in the history. */
                hist–>h_accept = 1;                  /* accept */
                return;                              /* done */
        }
        if (messbuffered(src, bc–>b_messnr))         /* is message buffered? */
                data = getmsg(src);                  /* yes, get it */
        else received = 0;                           /* not received the data yet */
   }
   if (g–>g_nextseqno == bc–>b_seqno && received) { /* accept it? */
        hist = hst_store(g, bc, data, n);            /* store new msg in history */
        hist->h_accept = 1;                          /* accept it */
        processrec(g);                               /* process history */
   } else if (g–>g_nextseqno < bc–>b_seqno ||
            (g–>g_nextseqno == bc–>b_seqno && !received)) {/* out of order? */
        if (received)  hst_store(g, bc, data, n);    /* yes, buffer it */
        ask_for_retransmission(g, g–>g_nextseqno);   /* ask for the missing msgs */
   }                                                 /* old message; discard */
}
```

Fig. 17. Algorithm for processing an incoming broadcast.

When a BC_BCAST arrives, the receiving kernel executes the procedure *broadcastreceive* (see Fig. 17). The kernel checks if the incoming message has user-supplied header and data. A BC_BCAST message with user-supplied header and data is a message sent following the PB method; otherwise the message is a short accept broadcast and the user-supplied header and data (received on a previous message) are buffered. If the sequence number is the one it expected, the message is stored in the history and processed by *processrec*. If the sequence number is not the expected one, the member has missed one or more broadcasts and asks the sequencer for retransmissions (BC_RETRANS). Out-of-

sequence broadcasts are buffered in the history, but the message is not processed, because the kernel is required to pass messages to the application in the correct order.

```
void processrec(g)
   struct group *g;                                  /* pointer to group state */
{
   struct member *src;                               /* pointer to original sender */
   struct hist *hist;                                /* pointer into the history */

   for(hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)];/* set pointer into history */
      hist–>h_accept;                                /* is the message accepted? */
      hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)]) {/* set pointer to next entry */
         src = g–>g_member + hist–>h_bc.b_cpu;        /* set source */
         g–>g_nextseqno++;                            /* accept it */
         g–>g_nexthistory++;                          /* increase next history */
         src–>m_messid = hist–>h_bc.b_messnr;         /* remember last used message id */
         if (src != g–>g_me)                          /* did I send the message? */
                  src–>m_expect = MAX(src–>m_expect, hist–>h_bc.b_expect);
         if ((g–>g_flags & FL_SENDING) && hist–>h_bc.b_cpu == g–>g_index) {
                  /* I sent this message; unblock sending thread. */
                  g–>g_flags &= ~FL_SENDING;          /* switch flag off */
                  unblock(g);                         /* unblock sending thread */
         }
         if (IAMSILENT(g)) sendstate(g);              /* silent for a long time? */
   }
}
```

Fig. 18. Function *processrec* used in Fig. 17.

Processrec inspects the history buffer to see if the next expected message has been stored (see Fig. 18). If so, it increases *g_nextseqno* and updates its member information of the sender. If the sender of the message is a member at the receiving kernel, then the sending member is unblocked and the *SendToGroup* returns successfully.

The sequencer uses the piggybacked acknowledgements contained in the messages from the members to determine which history buffers to free. The assumption is that a member will always send a message before the history fills up. This assumption need not be true, depending on the communication patterns of the applications. For example, a member may send a message that triggers another member to send messages. If this member misses the message, the system may very well become deadlocked. To prevent this from happening, each member kernel sends a BC_STATE message after receiving a certain number of messages without sending any. This is effectively a logical timer; a real timer could also be used, but this would be less efficient. The *b_expect* field in the BC_STATE message informs the sequencer which messages have been delivered by the sender of the message.

If the sequencer runs out of history buffers and has not received enough BC_STATE messages to make the decision to free history buffers, it can explicitly ask members which messages they have received. It does so by sending a BC_SYNC message. Members respond to this message with a BC_STATE message.

5.3. Protocol for Recovery

In the previous section, we assumed that none of the members ever failed. Now we will discuss the algorithms that are executed when a member or the sequencer fails.

Failures are detected by sending a BC_ALIVEREQ message to a kernel that has not been heard from in some time, and waiting for a reply. If, after a number of retries no BC_ALIVE message comes back, the enquiring kernel assumes that the destination has failed and initiates recovery. Picking the right number of retries is tricky. If the number is too low, a kernel may decide that another member has failed while in reality the other group member was just busy doing other things. If the number is too high, it can take a long time before a failure is detected.

Once a kernel has decided that another kernel has failed, it enters a recovery mode that causes subsequent calls to *ReceiveFromGroup* from local members to 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 all the surviving members that can communicate 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 the required number of members.

The protocol to recover from member crashes is based on the invitation protocol described by Garcia-Molina [Garcia-Molina 1982]. We will briefly describe the protocol for recovery and the extensions to support total ordering, but for the exact details and the correctness proof see [Garcia-Molina 1982]. The invitation protocol 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* becomes a coordinator and invites other members to join the new group by sending a BC_REFORMREQ message. If a member is alive and it is not a coordinator, it responds with a BC_VOTE message containing 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 are 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). Once the coordinator is up-to-date, it checks to see if the complete history is replicated on all remaining members. If one of the members does not have the complete history, the coordinator sends it the missing messages. Once the new group is up-to-date, it builds a BC_RESULT message containing information about the new group: the size of the new group, the members in the new group, the new sequencer (itself), a new network address for the group, and the new *incarnation* number of the group.

The network address and incarnation number are included to make sure that messages directed to the old group will not be accepted by the new group. It stores the BC_RESULT message in its history and broadcasts it to all members. When a member receives the BC_RESULT message, it updates the group information, sends an acknowledgement (BC_RESULTACK) to the coordinator, and enters normal operation. The coordinator enters normal operation after it has received a BC_RESULTACK 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, for example, due to a network partition, and did not take part in the recovery will still use an old incarnation number when sending a message to the new group. These messages 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 back to life 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 Figure 19 the recovery protocol is illustrated for a simple case where the resilience degree is zero. In Figure 19(a) a possible start of phase 1 is depicted. Members 0, 1, and 2 simultaneously start 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 Figure 19(b), the end of phase 1 has been 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 Figure 19(c), the end of phase 2 has been 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 BC_RESULT message.

The interaction between ordering, recovery, and reliability is complex. The key to understanding the correctness of the protocol is (1) that only the sequencer orders messages and (2) that a new sequencer starts only after all the members in its group have received all the messages that were sent successfully before a failure happened.

Consider a group with $r = 2$ (r is the resilience degree). Now the protocol has to guarantee that even if 2 members fail the remaining members will receive in the same order all message sent successfully (i.e., *SendToGroup* returned successfully) before the failure. With $r = 2$ the protocol for sending a message guarantees that the sequencer, processors P_1, and P_2 have a copy of the messages before any *SendToGroup* returns successfully. With $r = 2$ a message is only accepted in the total order if two other processors besides the sequencer have received the message (including its sequence number). In addition, the processors P_1 and P_2 may have a number of messages (including sequence number) for which they have sent the acknowledgements but for which they have missed the BCAST; other members (e.g., the sequencer) may have received the BCAST and may have delivered the messages.

Now suppose the worst case happens: the sequencer fails. As soon as a member detects this

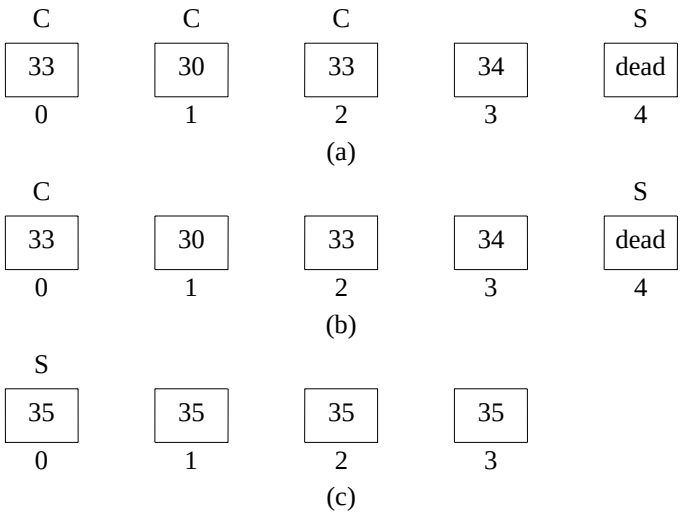


Fig. 19. A possible recovery for a group of size 5 after a crash of member 4. S is a sequencer. C is a coordinator. The number in the box is the sequence number of the last message received. The number below the box is the member id. (a) Shows a possible start of phase 1. (b) Shows the start of phase 2. In (c) the recovery has been completed.

failure, the recovery protocol is started and all members will stop trying to send application messages. P_1 or P_2 will be elected as coordinator, as they have the lowest rank and they have received all messages successfully sent (they have sent the acknowledgements allowing the sequencer to accept the message). After the coordinator has determined which processors are alive, it feeds all the surviving members the messages that it received before the failure was detected. Now the group is restarted and the coordinator sends every surviving member a message saying that it can start sending. The coordinator does not order any new messages until it has received an acknowledgement from all the members in the rebuilt group. Once the coordinator has received an acknowledgement from every member, it becomes the new sequencer and starts ordering messages. At this point every member has received all the messages that have been sent before the failure and also has received them in the same total order.

If another processor fails during recovery, the recovery starts from scratch. One processor of the sequencer, P_1, and P_2 has a copy of the messages that have been successfully sent before the first failure and it can bring the remaining members up-to-date. Other complex interactions of failures are possible, but the key observation is that the protocol proceeds in lockstep after a failure; before proceeding after a failure, all the remaining members are brought up-to-date.

6. Implementation and Performance

Group communication is implemented as part of the Amoeba kernel. We will now describe how group communication is integrated with the Amoeba kernel and give detailed performance figures.

6.1. Implementation

The communication system in the kernel consists of 3 layers (see Fig. 20). The top layer implements the protocols for group communication and RPC. The protocols described in the previous section are implemented here.

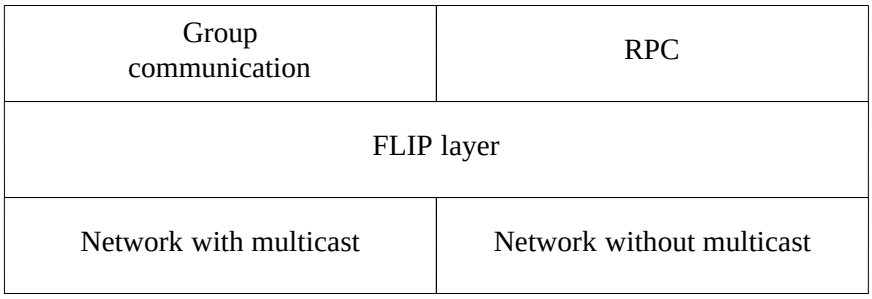


Fig. 20. Communication layers in the Amoeba kernel.

Both the RPC and group communication modules use the Fast Local Internet Protocol (FLIP) to send messages [Kaashoek et al. 1993b]. FLIP is a connectionless (datagram) protocol, roughly analogous to IP [Postel 1981], but with increased functionality. For the experiments performed in this section we could have used multicast-IP [Deering 1988; Deering and Cheriton 1990] instead of FLIP, but FLIP has other properties that makes it attractive for distributed computing. One of the major differences between IP and FLIP is that IP addresses identify a host while FLIP addresses identify a process or a group of processes. This simplifies, for example, the implementation of process migration and of group communication. FLIP is specifically designed to support a high-performance group communication and RPC protocol rather than support byte-stream protocols like TCP or OSI TP4. FLIP treats the ability of a network to send multicast messages as an optimization over sending *n* separate point-to-point messages.

6.2. Performance

The measurements were taken on a collection of 30 MC68030s (20 Mhz) connected by a 10 Mbit/s Ethernet. All processors were on the same Ethernet and were connected to the network by Lance chip interfaces (manufactured by Advanced Micro Devices). The protocols also work for network configurations in which members are located on different networks; FLIP will ensure that the messages are routed appropriately. We measured the case in which the members are located on the same network, as most network traffic is local and the results reported in the literature on comparable experiments are also for this setup. Thus, in the experiments all the members can be reached by sending one multicast packet. The machines used in the experiments were able to buffer 32 Ethernet

packets before the Lance overflowed and dropped packets. Each measurement was done 10,000 times on an almost quiet network. The size of the history buffer was 128 messages. The experiments measured failure-free performance.

Most experiments were executed with messages of size 0 bytes, 1 Kbyte, 4 Kbyte, and 8,000 byte. The last size was chosen to reflect a fundamental problem in the implementation. In principle, the group communication protocols can handle messages of size 8,000 or larger, but lower layers in the kernel prohibit measuring the communication costs for these sizes. Messages larger than a network packet size have to be fragmented into multiple packets. To prevent a sender from overrunning a receiver, flow control has to be performed on messages consisting of multiple packets. For point-to-point communication many flow control algorithms exists [Tanenbaum 1989], but it is not immediately clear how these should be extended to multicast communication. Some recent progress has been made in this area [Amir et al. 1992], but the results are not widely applicable yet. The measurements in this section therefore do not include the time for flow control and we have used a reasonable, but arbitrary, upper bound to the message size.

The first experiment measures the delay for the PB method with $r = 0$. In this experiment one process continuously broadcasts messages of size 0 byte, 1 Kbyte, 4 Kbyte, and 8,000 byte to a group of processes (the size of the message excludes the 116† bytes of protocol headers). All members continuously call *ReceiveFromGroup*. This experiment measures the delay seen from the sending process, between calling and returning from *SendToGroup*. The sending process runs on a different processor than the sequencer. Note that this is not the best possible case for our protocol, since only one processor sends messages to the sequencer (i.e., no acknowledgements can be piggybacked by other processors).

The results of the first experiment are depicted in Figure 21. For a group of two processes, the measured delay for a 0-byte message is 2.7 msec. Compared to the Amoeba RPC on the same architecture, the group communication is only 0.1 msec slower than the RPC. For a group of 30 processes, the measured delay for a 0-byte message is 2.8 msec. From these numbers, one can estimate that each node adds 4 μsec to the delay for a broadcast to a group of 2 nodes. Extrapolating, the delay for a broadcast to a group of 100 nodes should be 3.2 msec. Sending an 8,000-byte message instead of a 0-byte message adds roughly 20 msec. Because the PB method is used in this experiment, this large increase can be attributed to the fact that the complete message goes over the network twice.

Figure 22 breaks down the cost for a single 0-byte *SendToGroup* to a group of size 2, using the PB method. Both members call ReceiveFromGroup to receive messages. To reflect the typical usage of the group primitives, ReceiveFromGroup is called by another thread than *SendToGroup*. Most of

---

† 116 is the number of header bytes: 14 bytes for the Ethernet header, 2 bytes flow control, 40 bytes for the FLIP header, 28 bytes for the group header, and 32 bytes for the Amoeba user header. The Amoeba user header is only sent once.

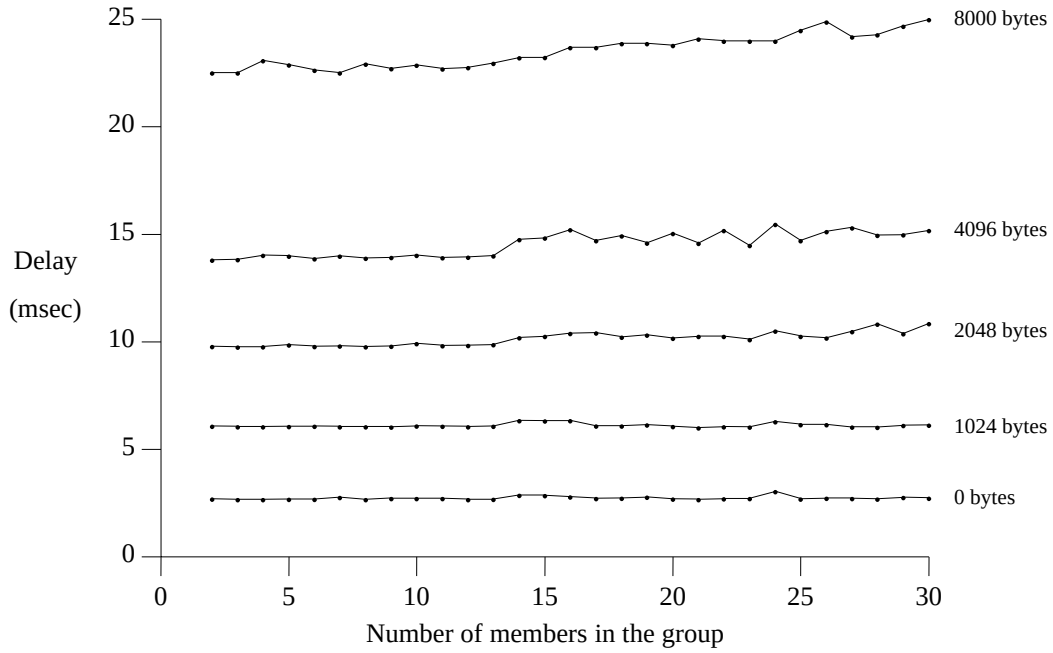


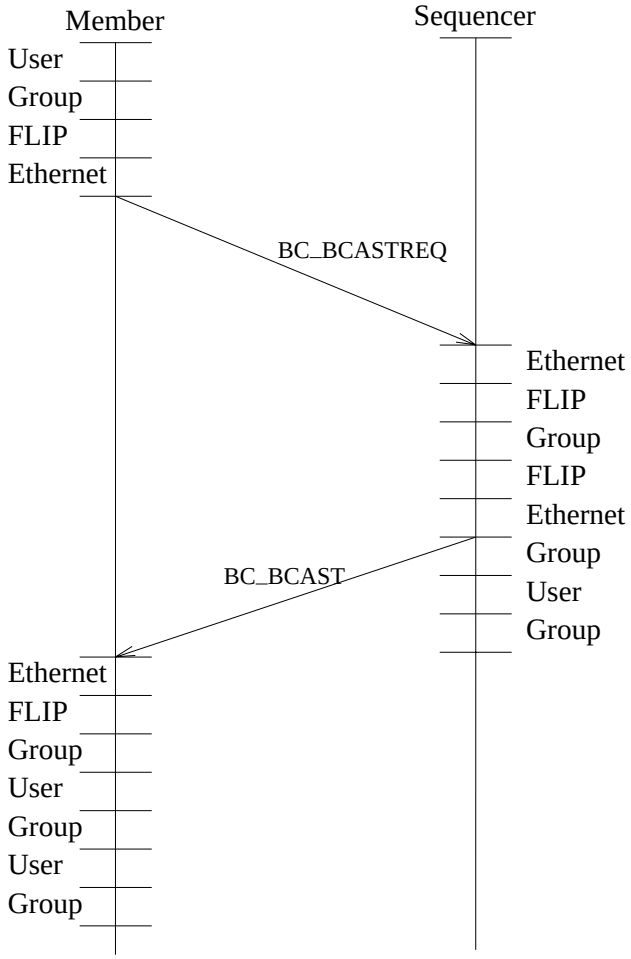
Fig. 21. Delay for 1 sender using PB method.

the time spent in user space is the context switch between the receiving and sending thread. The cost for the group protocol itself is 740 μsec.

The results of the same experiment but now using the BB method are depicted in Figure 23. The result for sending a 0-byte message is, as can be expected, similar. For larger messages the results are dramatically better, since in the BB method the complete message only goes over the network once. At first sight, it may look as if the BB method is always as good as or better than the PB protocol. However, this is not true. From the point of view of a single sender there is no difference in performance, but for the receivers other than the sequencer there is. In the PB protocol they are interrupted once, while in the BB protocol they are interrupted twice.

The next experiment measures the throughput of the group communication. In this experiment all members of a given group continuously call *SendToGroup*. We measure both for the PB method and the BB method how many messages per second the group can deal with. The results are depicted in Figure 24 and Figure 25. The maximum throughput is 815 0-byte messages per second. The number is limited by the time that the sequencer needs to process a message. This time is equal to the time spent taking the interrupt plus the time spent in the driver, FLIP protocol, and broadcast protocol. On the 20-MHz 68030, this is almost 800 μsec, which gives an upper bound of 1250 messages per second. This number is not achieved, because the member running on the sequencer must also be scheduled and allowed to process the messages.

The throughput decreases as the message size grows, because more data have to be copied. A



(a)

Layer	Time (μsec)
User	514
Group	740
FLIP	570
Ethernet	916

(b)

Fig. 22. (a) A break down of the cost in μsec of a single *SendToGroup* - *ReceiveFromGroup* pair. The group size is 2 and the PB method is used.
(b) The time spent in the critical path of each layer. The Ethernet time is the time spend on the wire plus the time spent in the driver and taking the interrupt.

receiver must copy each message twice: once from the Lance interface to the history buffer and once from the history buffer to user space. In the PB method, the sequencer must copy the message three times: one additional copy from the history buffer to the Lance interface to broadcast the message. (If our Lance interface could have sent directly from main memory, this last copy could have been avoid-

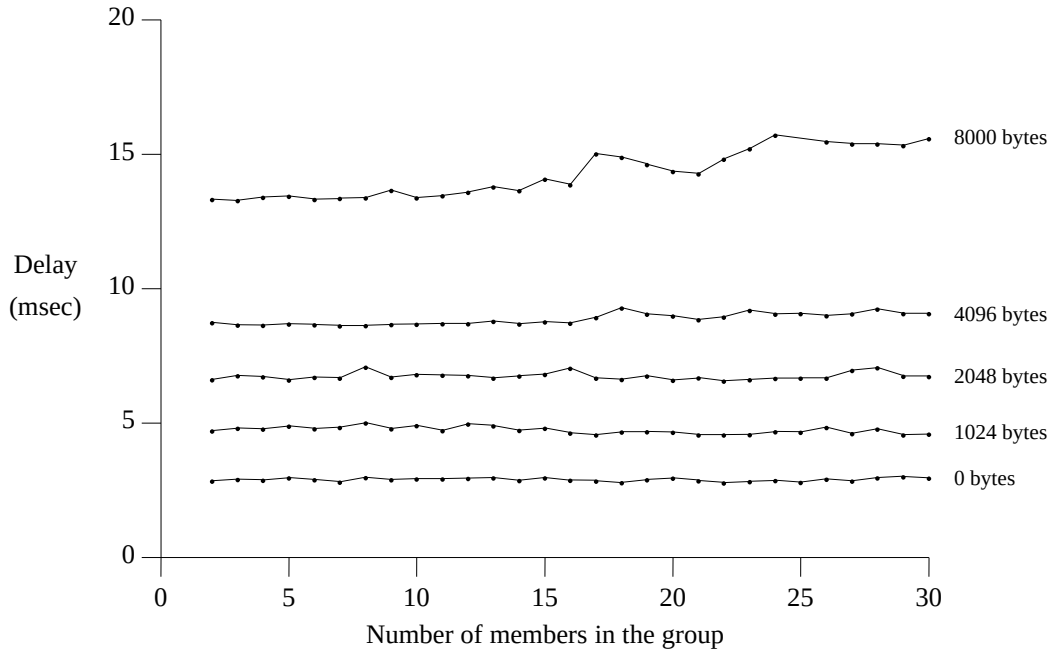


Fig. 23. Delay for 1 sender using the BB method.

ed.) If Amoeba had support for sophisticated memory management primitives like Mach [Young et al. 1987] the second copy from the history buffer to user space could also have been avoided; in this case one could map the page containing the history buffer into the user's address space.

For messages of size 4 Kbyte and larger, the throughput drops more. (For some configurations we are not able to make meaningful measurements at all.) This comes from the fact that our Lance configuration can buffer only 32 Ethernet packets, each with a maximum size of 1514 bytes. This means that the sequencer starts dropping packets when receiving 11 complete 4 Kbyte messages simultaneously. (If our system had been able to buffer more packets, the same problem would have appeared at some later point. The sequencer will need more time to process all the buffered packets, which will at some point result in timeouts at the sending kernel and in retransmissions.) The protocol continues working, but the performance drops, because the protocol waits until timers expire to send retransmissions. The same phenomenon also appears with groups larger than 16 members and 2-Kbyte messages.

Another interesting question is how many disjoint groups can run in parallel on the same Ethernet without influencing each other. To answer this question we ran an experiment in which a number of groups of the same size operated in parallel and each member of each group continuously called *SendToGroup*. We ran this experiment for group sizes of 2, 4, and 8 and measured the total number of 0-byte broadcasts per second (using the PB method). The experiment measures, for example, for two groups with 2 members the total number of messages per second that 4 members together succeeded in sending, with each member being member of one group and running on a separate proces-

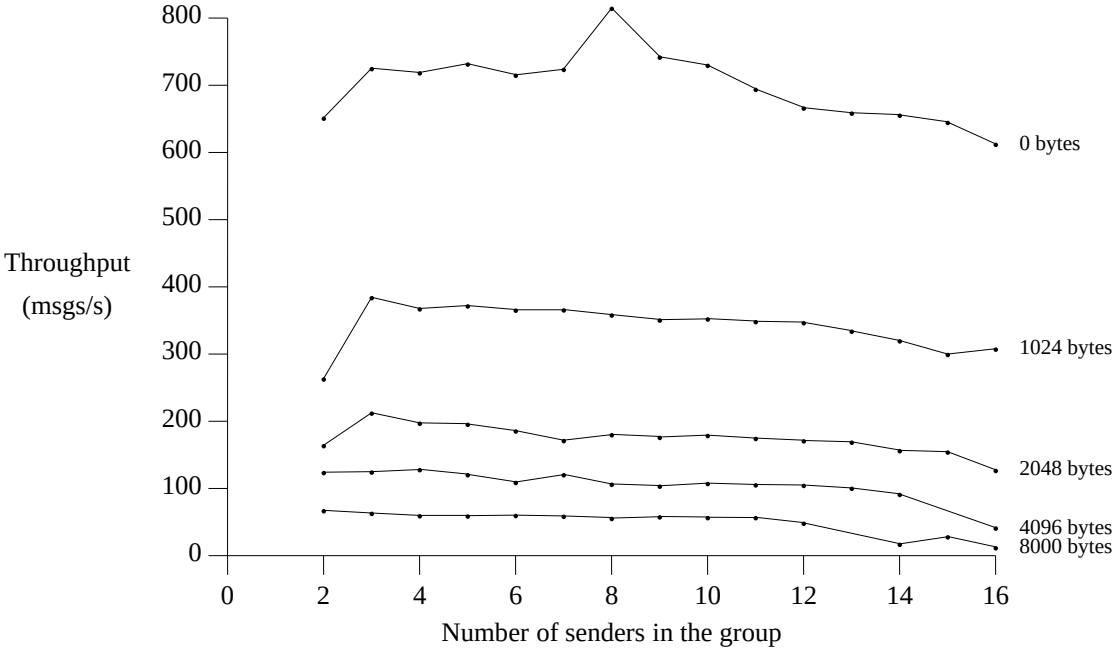


Fig. 24. Throughput for the PB Method. The group size is equal to the number of senders.

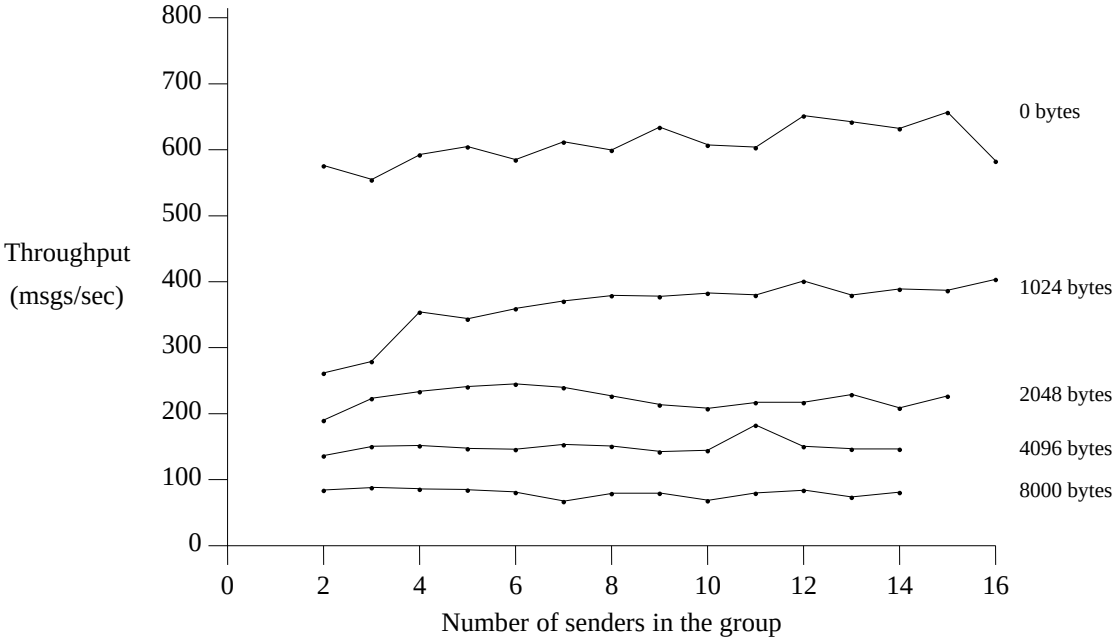


Fig. 25. Throughput for the BB Method. The group size is equal to the number of senders.

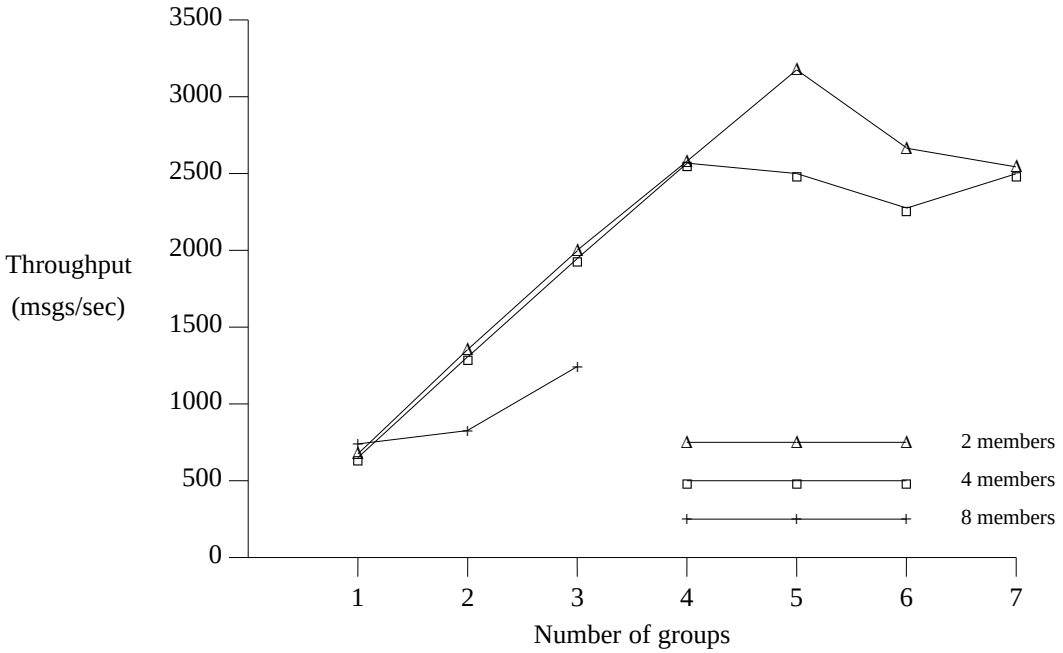


Fig. 26. Throughput for groups of 2, 4, and 8 members running in parallel and using the PB method. We did not have enough machines available to measure the throughput with more groups with 8 members.

sor. The results are depicted in Figure 26. The maximum throughput is 3175 broadcasts per second when 5 groups of size 2 are broadcasting at maximum 0-byte message throughput (this corresponds to at least 736,600 bytes per second; $3175 * 2 * 116 = 736,600$). When another group is added the throughput starts dropping due to the number of collisions on the Ethernet. This is also the case for the poor performance of groups of size 8.

The final experiment measures the delay of sending a message with $r > 0$. Figure 27 and Figure 28 depict the delay for sending a message with *resilience degree*s from one to 15. As can be expected, sending a message with a higher r scales less well than sending with a degree of 0. In this case, the number of FLIP messages per reliable broadcast sent is equal to $3 + r$ (assuming no packet loss). Also, when using large messages and a high resilience degree, our hardware starts to miss packets. For these settings we are not able to make meaningful measurements.

The delay for sending a 0-byte message to a group of size two with a resilience degree of one is 4.2 msec. For a group of size 16 with a resilience degree of 15, the measured delay is 12.9 msec. This difference is due to the 14 additional acknowledgements that have to be sent. Each acknowledgement adds approximately 600 μsec.

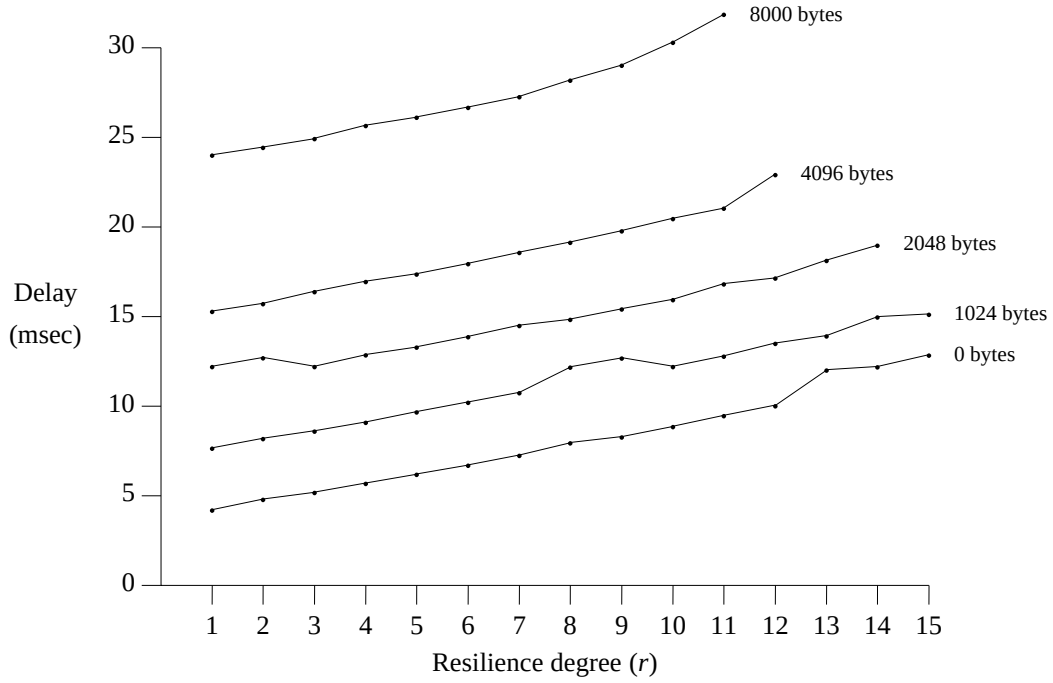


Fig. 27. Delay for 1 sender with different rs using PB method. Group size is equal to $r + 1$.

7. Comparison with Related Work

In this section, we will compare our reliable broadcast protocol with other protocols and our system with other systems that provide broadcast communication. Figure 29 summarizes the results. In comparing protocols, several points are of interest. The 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 messages needed to broadcast the message. The second is the semantics of sending a broadcast message. There are three aspects: reliability, ordering, and fault tolerance. Although fault tolerance is an aspect of reliability, we list it as a separate aspect. The 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) or requires synchronized clocks. For a bibliography of these papers we refer the reader to [Chanson et al. 1989].

Let us look at each protocol in turn. Amir et al. (ADKM) describe a recently built system, called Transis, that supports a number of protocols with varying properties [Amir et al. 1992]. It offers membership protocols, basic multicast (reliable group communication without order), causal-ordered multicast, totally-ordered multicast, and safe multicast (i.e., it delivers a message after all ac-

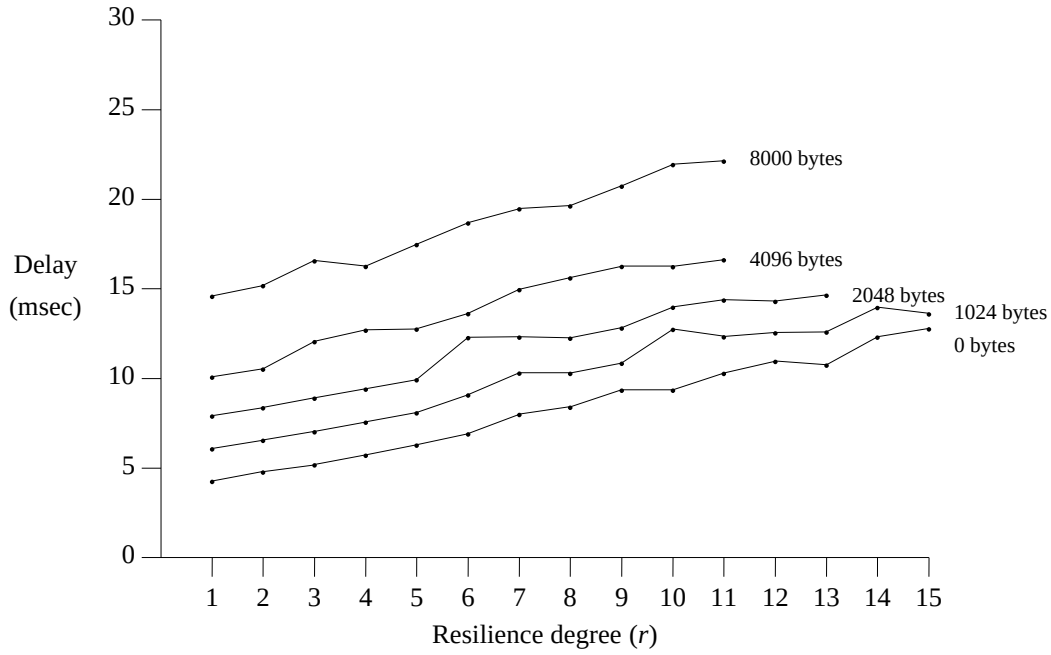


Fig. 28. Delay for 1 sender with different rs using BB method. Group size is equal to $r + 1$.

tive processors have acknowledged it). The approach used is similar to the one used in Psync (see below); the communication system builds a graph, in which the nodes are messages and the edges connect two messages that are directly dependent in the causal order. The services differ in the criteria that determine when to deliver a message to the application. In addition to the layering of broadcast services, Transis has two other distinctive properties. It provides support for groups to remerge after a partition and it implements multicast flow control. Preliminary performance results using broadcast (instead of multicast) show that the system performs well.

The protocols that are used in one of the first systems supporting ordered group communication are described by [Birman and Joseph 1987] 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 messages may be lost, in the $r = 0$ case. 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 includes primitives that provide a weaker ordering (e.g., a causal ordering).

Recently the protocols for Isis have been redesigned [Birman et al. 1991]. The system is now completely based on a broadcast primitive that provides causal ordering. The implementation of this primitive uses reliable point-to-point communication. The protocol for totally-ordered broadcast is based on causal broadcast. As in our protocol, a sequencer (a token holder in Isis terminology) is used to totally order the causal messages. Unlike our protocol, the token holder can migrate.

Depending if the sender holds the token, this scheme requires either one message or two messages, but each message possibly contains a sequence number for each member, while in our protocol the number of bytes for the protocol header is independent of the number of members. Thus in Isis, for a group of 1024 members, 4K bytes of data are possibly added to each message. Depending on the communication patterns, this data can be compressed, but in the worst case 4K is still needed. As an aside, the new version of Isis no longer supports a total ordering for overlapping groups. A reimplementation of Isis, called Horus, achieves very high performance by packing multiple messages in a single network packet, by avoiding major bottlenecks in the communication path, and by using multicast-IP [Van Renesse et al. 1992].

Chang and Maxemchuk describe a family of broadcast protocols [Chang and Maxemchuk 1984]. These protocols differ mainly in the degree of fault tolerance that they provide. Our protocol for $r = 0$ 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 also depends 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 and only a fraction bigger than 2 in the normal case.

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, but uses fewer messages than ours. 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; in ours only n. The actual implementation of their protocol uses physical broadcast for all messages and is restricted to a single LAN.

The group communication for the V system, described in [Cheriton and Zwaenepoel 1985], 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 returns successfully. 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 needs to know how the service is implemented. We do not think this is a good approach. If an unreplicated file service, for example, is re-implemented as a replicated file service to improve performance and to increase availability, it would mean that all client programs have to be changed. With our primitives, no change is needed in the client code.

Elnozahy and Zwaenepoel describe a broadcast protocol (EZ in Fig. 29) especially designed for replicated process groups [Elnozahy and Zwaenepoel 1992]. Like the CM protocols and like ours, it

Protocol	Performance		Semantics			Additional Hardware
	Delay	#Pkts	Reliable	Ordering	Fault-tolerance	
ADKM	≥1	1	Yes	No…Yes	$0 \ldots n-1$	No
BJ	2 Rounds	$2n$	Yes	Yes	$n-1$	No
BSS	2	$2n$	Yes	Yes	$n-1$	No
CM	$2 \ldots 2+n-1$	$2+\varepsilon$	Yes	Yes	$0 \ldots n-1$	No
CZ	2	$2 \ldots n$	No…Yes	No	No	No
EZ	2	2	Yes	Yes	$n-1$	No
LG	3 Phases	$1 \ldots 4n$	Yes	Yes	Yes	Yes
MMA	≥1	1	Yes	Yes	$n/2$	No
M	2	$2+\varepsilon$	Yes	Yes	Yes	Yes
NCN	2	$n+1$	Yes	No…Yes	$0 \ldots n-1$	No
PBS	≥1	1	Yes	Yes	$0 \ldots n-1$	No…Yes
TY	2	3	Yes	Yes	No…Yes	Yes
VRB	2 Rounds	$2n$	Yes	Yes	$n-1$	Yes
Ours	2	$2 \ldots 3+n-1$	Yes	Yes	$0 \ldots n-1$	No

Fig. 29. Comparison of different broadcast protocols. A protocol is identified by the first letters of the names of the authors. The group size is n. 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. In some cases, the performance may be worse, for example, for higher degrees of fault tolerance.

is based on a centralized site and negative acknowledgements. Unlike ours, it especially designed to provide a low delay in delivery of messages, while at the same time providing a high resilience degree. This goal is achieved by keeping an *antecedence graph* and adding to each message the incremental changes to this graph. By maintaining the antecedence graph this protocol does not need to send acknowledgements to confirm that the message is stored at r members. On the other hand, the application must potentially be rolled back when a processor fails.

The protocol described in [Luan and Gligor 1990] is one of the protocols that require additional hardware. In the LG protocol each member must be equipped with a disk. Using these disks the pro-

tocol can provide fault-tolerant ordered broadcasting, even if the network partitions. It uses a majority-consensus decision to agree 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 [Melliar-Smith et al. 1990]. 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 totally, but nevertheless there is a certain chance that messages are not totally-ordered. The MMA 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. With large groups, the delay is unacceptably large.

Montgomery [Montgomery 1978] coined the term *atomic broadcast* in an unpublished dissertation. The thesis describes the problem of reliable, totally-ordered multicast and proposes two solutions: one based on point-to-point communication and one based on broadcast. Both solutions are based on a centralized component that orders messages. To provide for fault tolerance the messages are always stored on stable storage. Another important difference between these two protocols and ours is that acknowledgements are not piggybacked. Instead, each node broadcasts once in a while a message saying which messages it has received, so that the central site can purge messages from its stable storage. No indication is given that the protocol was ever implemented, and no measurements are presented.

Navaratnam, Chanson, and Neufeld provide two primitives for reliable broadcasting [Navaratnam et al. 1988]. 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 on comparable hardware. Our current implementation does this in less than 4.8 msec ($r = 3$).

In [Peterson et al. 1989] a communication mechanism is described called *Psync*. In Psync a group consists of a fixed number of processes and is closed. Messages are causally ordered. A library routine provides a primitive for a total ordering. This primitive is implemented using a single causal message, but members cannot deliver a message immediately when it arrives. Instead, a number of messages from other members (i.e., at most one from each member) must be received before the total order can be established.

Another protocol that requires hardware support for reliable broadcasting is described in [Tseung and Yu 1990]. The TY protocol requires that at least three components be added to the net-

work: 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 a point-to-point 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 the TY protocol, the Retransmission Computer 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 acknowledgement from the Designated Recorder to the Retransmission Recorder is not needed in our protocol) and requires additional hardware. Furthermore, one computer (the Retransmission Computer) serves as 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 on a per group basis by having a separate sequencer for each group.

The last protocol that we consider which provides reliable broadcasting is described in [Verissimo et al. 1989]. The VRB protocol runs directly on top of the Medium Access Layer (MAC). Thus, the protocol is restricted to a single LAN, but on the other hand it allows for an efficient implementation. The protocol itself is based on the two phase commit protocol [Eswaran et al. 1976]. In the first phase, the message is broadcast. All receivers are required to send an acknowledgement indicating if they will accept the message. After the sender has received all acknowledgements, it broadcasts a message telling if the message can be delivered to the user application or not. The protocol assumes that the network orders packets and that there is a bounded transmission delay.

A somewhat related approach is Cooper's replicated RPC [Cooper 1985]. Although replicated RPC provides communication facilities for 1-to-n communication, it does not use group communication. Instead, it performs $n - 1$ RPCs. As it is not based on group communication, nor does it use multicast, we did not include it in the table. Replicated RPC, however, can be implemented using group communication [Wood 1993].

Another related approach is MultiRPC [Satyanarayanan and Siegel 1990]. From the programming point of view, MultiRPC behaves exactly the same as an ordinary RPC. However, instead of invoking one server stub, MultiRPC invokes multiple server stubs on different machines in parallel. Compared to performing $n - 1$ regular RPCs, MultiRPC is more efficient as the server stubs are executed in parallel. The authors also discuss preliminary results for sending the request message in a multicast packet to avoid the overhead of sending the requests n times in point-to-point packets. The replies on an RPC are sent using point-to-point communication and are processed sequentially by the

client machine. There is no ordering between two MultiRPCs and MultiRPC does not provide reliable communication in case one of the servers crashes.

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 listed in the table. 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, two messages are used: a point-to-point message to the sequencer and a broadcast message. 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, three messages per reliable broadcast are needed. In practice, with say 1 Mbyte of history buffers and 1 Kbyte-messages, there is room for 1024 messages. This means that the history buffer will rarely fill up and the protocol will actually average two messages per reliable broadcast. The delay for sending a message is equal to the time to send and receive a message from the sequencer. 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 (PB method).

If messages must be delivered in order and without error despite member crashes, the cost of the protocol increases. For resilience degree $r > 0$, each reliable broadcast takes $3 + r$ messages: one message for the point-to-point message to the sequencer, one broadcast message from the sequencer to all kernels announcing the sequence number, r short acknowledgements that are sent point-to-point to the sequencer, and one short accept message from the sequencer to all members. The delay increases. A message can only be accepted by the sequencer after receiving the message, broadcasting the message, and receiving r acknowledgements. However, the r acknowledgements will be received almost simultaneously. Thus, an increase in fault tolerance costs the application a decrease in performance. It is up to the application programmer to make the tradeoff.

Like some of the 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. If $r > 0$, it is likely that the sequencer will become a bottleneck sooner due to the r acknowledgements that it has to process. Under heavy load, one could try to piggyback these acknowledgements onto other messages, to make the protocol scale better.

8. Conclusion

We have identified 6 criteria that are important design issues for group communication: addressing, reliability, ordering, delivery semantics, response semantics, and group structure. We have discussed each of these criteria and the choices that have been made for the Amoeba distributed system. The Amoeba interface for group communication is simple, powerful, and easy to understand. Its main properties are:

- Reliable communication.

- Messages are totally-ordered per group.
- Programmers can trade performance against fault tolerance.

Based on our experience with distributed programming we believe that these properties are essential in building efficient distributed applications.

We have described in detail the group communication interface and its implementation. In addition, we have provided extensive performance measurements on 30 processors. The delay for a null broadcast to a group of 30 processes running on 20-MHz MC68030s connected by 10 Mbit/s Ethernet is 2.8 msec. The maximum throughput per group is 815 broadcasts per group. With multiple groups, the maximum number of broadcasts per second has been measured at 3175.

ACKNOWLEDGEMENTS

We would like to thank Henri Bal, Susan Flynn, and Wiebren de Jonge for their contributions to the broadcast protocol. In addition we would like to thank the anonymous referees, Mootaz Elnozahy, Robbert van Renesse, Kees Verstoep, Mark Wood, and Willy Zwaenepoel for providing comments on several drafts of this paper.

REFERENCES

Ahamad, M. and Bernstein, A. J., "An Application of Name Based Addressing to Low Level Distributed Algorithms," *IEEE Trans. on Soft. Eng.*, Vol. 11, No. 1, pp. 59-67, Jan. 1985.

Amir, Y., Dolev, D., Kramer, S., and Malki, D., "Transis: A Communication Sub-System for High Availability," *Proc. 22nd International Symposium on Fault-Tolerant Computing*, pp. 76-84, Boston, MA, June 1992.

Bal, H. E., "Programming Distributed Systems," Silicon Press, Summit, NJ, 1990.

Birman, K. P., "The Process Group Approach to Reliable Distributed Computing," *Comm. ACM*, Vol. 36, No. 12, pp. 36-53, Dec. 1993.

Birman, K. P., Cooper, R., Jospeh, T. A., Kane, K. P., Schmuck, F., and Wood, M., "Isis - A Distributed Programming Environment," User's Guide and Reference Manual, Cornell University, Ithaca, NY, June 1990.

Birman, K. P. and Joseph, T. A., "Reliable Communication in the Presence of Failures," *ACM Trans. Comp. Syst.*, Vol. 5, No. 1, pp. 47-76, Feb. 1987.

Birman, K. P., Schiper, A., and Stephenson, P., "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Comp. Syst.*, Vol. 9, No. 3, pp. 272-314, Aug. 1991.

Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, Feb. 1984.

Chang, J., “Simplifying Distributed Database Design by Using a Broadcast Network,” *Proc. ACM SIGMOD 1984 Annual Conference*, pp. 223-233, Boston, MA, June 1984.

Chang, J. and Maxemchuk, N. F., “Reliable Broadcast Protocols,” *ACM Trans. Comp. Syst.*, Vol. 2, No. 3, pp. 251-273, Aug. 1984.

Chanson, S. T., Neufeld, G. W., and Liang, L., “A Bibliography on Multicast and Group Communication,” *Operating Systems Review*, Vol. 23, No. 4, pp. 20-25, Oct. 1989.

Cheriton, D. R. and Skeen, D., “Understanding the Limitations of Causally and Totally Ordered Communication,” *Proc. Fourteenth Symposium on Operating System Principles*, pp. 44-57, Asheville, NC, Dec. 1993.

Cheriton, D. R. and Zwaenepoel, W., “Distributed Process Groups in the V kernel,” *ACM Trans. Comp. Syst.*, Vol. 3, No. 2, pp. 77-107, May 1985.

Cooper, E. C., “Replicated Distributed Programs,” *Proc. Tenth Symposium on Operating Systems Principles*, pp. 63-78, Orcas Islands, WA, Dec. 1985.

Cristian, F., “Understanding Fault-Tolerant Distributed Systems,” *Commun. ACM*, Vol. 34, No. 2, pp. 56-78, Feb. 1991.

Danzig, P. B., “Finite Buffers and Fast Multicast,” *Perf. Eval. Rev.*, Vol. 17, No. 1, pp. 79-88, May 1989.

Dechter, R. and Kleinrock, L., “Broadcast Communication and Distributed Algorithms,” *IEEE Trans. on Computers*, Vol. 35, No. 3, pp. 210-219, Mar. 1986.

Deering, S. E., “Host Extensions for IP Multicasting,” RFC 1112, SRI Network Information Center, Aug. 1988.

Deering, S. E. and Cheriton, D. R., “Multicast Routing in Datagram Internetworks and Extended LANs,” *ACM Trans. Comp. Syst.*, Vol. 8, No. 2, pp. 85-110, May 1990.

Elnozahy, E. N. and Zwaenepoel, W., “Replicated Distributed Processes in Manetho,” *Proc. 22nd International Symposium on Fault-Tolerant Computing*, pp. 18-27, Boston, MA, June 1992.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., “The Notion of Consistency and Predicate Locks in a Database System,” *Commun. ACM*, Vol. 19, No. 11, pp. 624-633, Nov. 1976.

Fischer, M. J., Lynch, N. A., and Paterson, M. S., “Impossibility of Distributed Consensus with One Faulty Process,” *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, Apr. 1985.

Frank, A. J., Wittie, L. D., and Bernstein, A. J., “Multicast Communication on Network Computers,” *IEEE Software*, Vol. 2, No. 3, pp. 49-61, May 1985.

Garcia-Molina, H., “Elections in a Distributed Computing System,” *IEEE Trans. on Computers*, Vol. 31, No. 1, pp. 48-59, Jan. 1982.

Gehani, N. H., "Broadcasting Sequential Processes," *IEEE Trans. on Soft. Eng.*, Vol. 10, No. 4, pp. 343-351, July 1984.

Gueth, R., Kriz, J., and Zueger, S., "Broadcasting Source-Addressed Messages," *Proc. Fifth International Conference on Distributed Computing Systems*, pp. 108-115, Denver, CO, 1985.

Hadzilacos, V. and Toueg, S., "Fault-Tolerant Broadcasts and Related Problems," in *Distributed Systems 2nd ed.*, ed. S. Mullender, Addison-Wesley, Reading, MA, 1993.

Hughes, L., "A Multicast Interface for UNIX 4.3," *Software−Practice and Experience*, Vol. 18, No. 1, pp. 15-27, Jan. 1988.

Hughes, L., "Multicast Response Handling Taxonomy," *Computer Communications*, Vol. 12, No. 1, pp. 39-46, Feb. 1989.

Joseph, T. A. and Birman, K. P., "Low Cost Management of Replicated Data in Fault-Tolerant Systems," *ACM Trans. Comp. Syst.*, Vol. 4, No. 1, pp. 54-70, Feb. 1986.

Kaashoek, M. F., "Group Communication in Distributed Computer Systems," Ph.D. Thesis, Vrije Universiteit, Amsterdam, 1992.

Kaashoek, M. F., Michiels, R., Bal, H. E., and Tanenbaum, A. S., "Transparent Fault-Tolerance in Parallel Orca Programs," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems III*, pp. 297-312, Newport Beach, CA, Mar. 1992.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., "An Efficient Reliable Broadcast Protocol," *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989.

Kaashoek, M. F., Tanenbaum, A. S., and Verstoep, K., "Using Group Communication to Implement a Fault-Tolerant Directory Service," *Proc. 13th International Conference on Distributed Computing Systems*, pp. 130-139, Pittsburgh, PA, May 1993a.

Kaashoek, M. F., Van Renesse, R., Van Staveren, H., and Tanenbaum, A. S., "FLIP: an Internetwork Protocol for Supporting Distributed Systems," *ACM Trans. Comp. Syst.*, Vol. 11, No. 1, pp. 73-106, Feb. 1993b.

Kung, H. T., "Gigabit Local Area Networks: a Systems Perspective," *IEEE Communications Magazine*, Vol. 30, No. 4, pp. 79-89, Apr. 1992.

Liang, L., Chanson, S. T., and Neufeld, G. W., "Process Groups and Group Communication: Classification and Requirements," *IEEE Computer*, Vol. 23, No. 2, pp. 56-68, Feb. 1990.

Luan, S. W. and Gligor, V. D., "A Fault-Tolerant Protocol for Atomic Broadcast," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 271-285, July 1990.

Marzullo, K. and Schmuck, F., "Supplying High Availability with a Standard Network File System," *Proc. Eighth International Conference on Distributed Computing Systems*, pp. 447-453, San Jose, CA, June 1988.

Melliar-Smith, P. M., Moser, L. E., and Agrawala, V., "Broadcast Protocols for Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 17-25, Jan. 1990.

Montgomery, W. A., "Robust Concurrency Control for a Distributed Information System," MIT/LCS/TR-207 (Ph.D. thesis), M.I.T., Cambridge, MA, Dec. 1978.

Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staveren, H., "Amoeba: a Distributed Operating System for the 1990s," *IEEE Computer*, Vol. 23, No. 5, pp. 44-53, May 1990.

Navaratnam, S., Chanson, S., and Neufeld, G., "Reliable Group Communication in Distributed Systems," *Proc. Eighth International Conference on Distributed Computing Systems*, pp. 439-446, San Jose, CA, June 1988.

Peterson, L. L., Buchholtz, N. C., and Schlichting, R. D., "Preserving and Using Context Information in IPC," *ACM Trans. Comp. Syst.*, Vol. 7, No. 3, pp. 217-246, Aug. 1989.

Postel, J., "Internet Protocol," RFC 791, SRI Network Information Center, Sep. 1981.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W., "Chorus Distributed Operating System," *Computing Systems*, Vol. 1, No. 4, pp. 305-370, 1988.

Satyanarayanan, M. and Siegel, E. H., "Parallel Communication in a Large Distributed Environment," *IEEE Trans. on Computers*, Vol. 39, No. 3, pp. 328-348, Mar. 1990.

Schneider, F. B., "Byzantine Generals in Action: Implementing Fail-Stop Processes," *ACM Trans. Comp. Syst.*, Vol. 2, No. 2, pp. 145-154, May 1984.

Schneider, F. B., "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 299-319, Dec. 1990.

Siegel, A., Birman, K., and Marzullo, K., "Deceit: a Flexible Distributed File System," *Proc. Usenix Summer Conference*, pp. 51-61, Anaheim, CA, June 1990.

Tanenbaum, A. S., "Computer Networks 2nd ed.," Prentice-Hall, Englewood Cliffs, NJ, 1989.

Tanenbaum, A. S., Kaashoek, M. F., and Bal, H. E., "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, Vol. 25, No. 8, pp. 10-19, Aug. 1992.

Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G., Mullender, S. J., Jansen, A., and Van Rossum, G., "Experiences with the Amoeba Distributed Operating System," *Commun. ACM*, Vol. 33, No. 12, pp. 46-63, Dec. 1990.

Tseung, L. C. N. and Yu, K-C., “The implementation of Guaranteed, Reliable, Secure Broadcast Networks,” *1990 ACM Eighteenth Annual Computer Science Conference*, pp. 259-266, Washington D.C., Feb. 1990.

Van Renesse, R., “Causal Controversy at Le Mont St. Michel,” *Operating Systems Review*, Vol. 27, No. 2, pp. 44-53, Apr. 1993.

Van Renesse, R., Birman, K. P., Cooper, R., Glade, B., and Stephenson, P., “Reliable Multicast between Microkernels,” *Proc. of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, Apr. 1992.

Verissimo, P., Rodrigues, L., and Baptista, M., “AMp: a Highly Parallel Atomic Multicast Protocol,” *Proc. SIGCOMM 89*, pp. 83-93, Austin, TX, Sep. 1989.

Wood, M. D., “Replicated RPC Using Amoeba Closed Group Communication,” *Proc. of the 13th Conference on Distributed Computing Systems*, pp. 499-507, Pittsburgh, PA, May 1993.

Young, M., Tevenian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., “Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System,” *Proc. Eleventh Symposium on Operating Systems Principles*, pp. 63-67, Austin, TX, Nov. 1987.