AN EFFICIENT RELIABLE BROADCAST PROTOCOL

M. Frans Kaashoek
Andrew S. Tanenbaum
Susan Flynn Hummel
Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Email: kaashoek@cs.vu.nl

ABSTRACT

Many distributed and parallel applications can make good use of broadcast communication. In this paper we present a (software) protocol that simulates reliable broadcast, even on an unreliable network. Using this protocol, application programs need not worry about lost messages. Recovery of communication failures is handled automatically and transparently by the protocol. In normal operation, our protocol is more efficient than previously published reliable broadcast protocols. An initial implementation of the protocol on 10 MC68020 CPUs connected by a 10 Mbit/sec Ethernet performs a reliable broadcast in 1.5 msec.

1. INTRODUCTION

Most current distributed operating systems are based on remote procedure call (RPC) [Birrell and Nelson 1984]. For many distributed and parallel applications, however, this sender-to-receiver-and-back communication style is inappropriate. What is frequently needed is *broadcasting*, in which an arbitrary one of the n user processes sends a message to the other $n - 1$ processes. Although broadcasting can always be simulated by sending $n - 1$ messages and waiting for the $n - 1$ acknowledgements, this algorithm is slow, inefficient, and wasteful of network bandwidth. In this paper we discuss a new protocol that allows 100% reliable broadcasting to be implemented on unreliable networks in only two messages per broadcast.

Before getting into the protocol, let us first look at two example applications in which broadcasting is a more suitable paradigm than RPC. First consider the traveling salesman problem [Lawler and Wood 1966]. In this problem, the computer is given a starting location and a list of cities to be visited. The idea is to find the shortest path that visits each city exactly once. When this problem is solved on a distributed system, processes can search different paths in parallel [Bal et al. 1987]. When a process finds a new path that is shorter than any path found so far, it wants to broadcast this path to all other processes, so that they will not waste time working on partial paths that are longer than the best known complete path. Here, a reliable broadcast from one process to all the others is a more appropriate model than RPC.

Another example is a system that replicates data structures (e.g. data base records) at multiple sites, to enhance fault tolerance and reduce access time. When a process needs to change a distributed data structure, it must update or invalidate all other copies. Again here, broadcasting is a better fit than RPC. Many other distributed applications in which some kind of global state is needed are also candidates for using broadcasting.

Interestingly enough, broadcast communication is provided by many kinds of networks, including LANs, geosynchronous satellites, and cellular radio systems [Tanenbaum 1989]. Thus, the hardware often supports the broadcasting that the applications need. It is the operating system that gets in the way. The protocol described in this paper can easily be integrated into an operating system so that the hardware support for broadcasting can be made available to application programs.

The rest of the paper is structured as follows. In Section 2, we describe the broadcast protocol itself. In Section 3, we describe our initial implementation of the protocol and present some measurements of its performance. In Section 4, we compare our protocol to several other published protocols. Finally, in Section 5, we give our conclusions.

2. THE BROADCAST PROTOCOL

We are concerned with distributed systems, each of which consists of a group of n processes that communicate through a broadcast network. For simplicity, we assume that each process runs on a separate node and that the system runs a single application. The generalization to multiprogramming and multiple applications is straightforward.

F1: [C1: circle radius 0.4 "Application" B1: [box "Kernel"] with .n at C1.s - (0,0.5) line from B1.s to B1.e arrow " sequencer" "disabled" from B1.se + (0.6, 0.1) to B1.se + (-0.05, 0.1) C2: [circle invis "●●●"] with .c at C1.c + (1.5, 0) B2: [box invis "●●●"] with .n at C2.s - (0,0.5) C3: circle radius 0.4 "Application" with .c at C2.c + (1.0, 0) B3: [box "Kernel"] with .n at C3.s - (0,0.5) line from B3.s to B3.e arrow " sequencer" "disabled" from B3.se + (0.6, 0.1) to B3.se + (-0.05, 0.1) C4: circle radius 0.4 "Application" with .c at C3.c + (1.5, 0) B4: [box "Kernel"] with .n at C4.s - (0,0.5) line from B4.s to B4.e arrow " sequencer" "enabled" from B4.se + (0.6, 0.1) to B4.se + (-0.05, 0.1)] arrow <-> from F1.C1.s to F1.B1.n arrow <-> from F1.C3.s to F1.B3.n arrow <-> from F1.C4.s to F1.B4.n arrow <-> down from F1.B1.s arrow <-> down from F1.B3.s arrow <-> down from F1.B4.s line "Broadcast network" below from F1.sw - (0,0.5) to F1.se - (0,0.5)

Fig. 1. System structure. Each node runs a kernel and an 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.

Each node runs a kernel and an application process (see Fig. 1). The kernel handles all the communication, including the broadcasting. The broadcast protocol to be described shortly runs inside the kernel. Any of the application processes can, at any instant, decide to send a broadcast message to the other $n - 1$ application processes. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communications, lost packets, and finite buffers.

Furthermore, if two application processes, on two different machines, simultaneously broadcast two messages, A and B respectively, then the kernel must guarantee that either everybody receives A first and then B or everybody receives B first and then A, but not some random mixture where some processes get A first and others get B first. By making the broadcasts both reliable and atomic in this way the user semantics become much simpler and easier to understand.

Both the kernel and the application processes may be structured internally, for example in layers, but the internal structure will not concern us further in this paper.

The basic reliable broadcast protocol works as follows. When an application process wants to broadcast a message, M, it hands the message to its kernel (e.g., using a system call). The kernel then encapsulates M in an ordinary point-to-point message and sends it to a special kernel called the *sequencer*. The sequencer's node contains the same hardware and kernel as all the others. The only difference is that a flag in the kernel tells it to process messages differently. If the sequencer should crash, the protocol can easily be extended to provide for the election of a new sequencer on a different node.

When the sequencer receives the point-to-point message containing M, it allocates the next sequence number, s and broadcasts a packet containing M and s. Thus all broadcasts are issued from the same node, by the sequencer. Assuming that no packets are lost, it is easy to see that if two application processes simultaneously want to broadcast, one of them will reach the sequencer first and its message will be broadcast to all the other nodes first. Only when that broadcast has been completed will the other broadcast be started. The sequencer provides a global ordering in time. In this way, we can easily guarantee the atomicity of broadcasting.

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

The kernel then sends a special point-to-point message to the sequencer asking it for copies of the missing message (or messages, if several have been missed). To be able to reply to such requests, the sequencer stores old broadcast messages in its *history buffer*. The missing messages are sent point-to-point to the process requesting them.

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

The protocol has several ways of letting the sequencer discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message. In this way, the sequencer can maintain a table, indexed by node number, showing that node i has received all broadcast messages 0 up to T_i, and perhaps more. At any moment, the sequencer 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 sequencer knows that everyone has received broadcasts 0 through 6, so they can be deleted from the history buffer.

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

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

Before looking at the detailed algorithms used to run the protocol, let us briefly summarize its properties. First, as far as application processes are concerned, broadcasts are reliable. When a process tells the kernel to broadcast a message, that process does not have to worry about what happens if a packet is lost. Like using TCP/IP, OSI transport class 4, or writing on UNIX† pipes, the sender can just assume that everything works. Making it work is the system's responsibility, not the user's. This property greatly simplifies distributed programming.

Second, broadcasts are atomic. Even if two application processes simultaneously decide to broadcast, one of them will go first, and then the other. They will not be interleaved. If some node misses the first one, it will see that it has missed one, when it gets the second one, and fetch that one immediately. Under no circumstances are broadcast messages given from the kernel to the application in the wrong order, not even when some messages are lost.

Third, the overhead is extremely low. To send a broadcast, only two packets are needed (assuming that a message fits in a single packet), one point-to-point packet from the sender to the sequencer and one broadcast packet from the sequencer to everyone. Only n interrupts are generated, one for the point-to-point packet, and $n - 1$ for the broadcast packets. This is an enormous improvement over RPC, in which $3(n - 1)$ packets (a request, a reply, and an acknowledgement for the reply per RPC) must be sent and $3(n - 1)$ interrupts handled to achieve reliable broadcasting; to achieve atomicity, even more packets are needed.

2.1. Normal operation

In this section, we will describe in detail how the sender, sequencer, and receivers behave during normal operation. Fig. 2 shows the data structures used by the protocol.

When a node wants to broadcast a message, it sends the sequencer a data message containing *LastSeqReceived*, *MessageId* and its processor number (see Fig. 3). *LastSeqReceived* informs the sequencer which messages have been received by the sender. In essence, *LastSeqReceived* is a piggybacked acknowledgement for all messages up to and

† UNIX is a Registered Trademark of AT&T in the USA and other countries.

```
/* Variables maintained on all machines. */
unsigned int LastSeqReceived;                  /* sequence number of the last broadcast seen */
unsigned int MessageId;                        /* unique id for next broadcast message */

/* Variables maintained by the sequencer */
unsigned int NextSeqToUse;                     /* next sequence number to use */
unsigned int SequenceNr[NUMBER_OF_CPUS];       /* biggybacked acks from the other kernels */
unsigned int MessageNr[NUMBER_OF_CPUS];        /* id of last broadcast from each kernel */

/* A message consists of pointers to a header and some data */
struct message *History[HISTORY_SIZE];         /* old broadcasts stored here */

/* Message header, showing the fields used */
struct header {
    unsigned int type;                         /* DATA, BROADCAST, RETRANS, ... */
    unsigned int SequenceNr;                   /* sequence number */
    unsigned int MessageNr;                    /* unique message id for detecting duplicates */
    unsigned int SenderID;                     /* sender's node number */
    unsigned int DestID;                       /* destination node number */
};
```

Fig. 2. Declarations.

including *LastSeqReceived*. *MessageId* and the node number ensure that each message is unique, and make it possible to recognize duplicates of a message. After sending the message, *MessageId* is incremented and the sending process is blocked and a timer can be started. If the broadcast fails to arrive, the timer will expire, and the message will be sent to the sequencer again. The combination of *MessageId* and the node number uniquely identifies each message. When the sequencer gets a request to broadcast a message, it checks to see if it has already done the broadcast. If so, it does not do it a second time.

The two arrays *SequenceNr* and *MessageNr* are maintained by the sequencer. The former is the table of piggybacked acknowledgements that allows it to figure out which history buffer entries can be safely removed. The latter gives the last *MessageId* received from each node and is used for detecting duplicates generated by timeouts. The *message* structure is the history buffer, and *HISTORY_SIZE* determines how big it is. (We assume that there is an upper bound to the size of a message.)

A message header contains 5 fields (see Fig. 2). The *type* field tells the kind of a message:

Type	From	To	Function
DATA	node	sequencer	Data to be broadcast
BROADCAST	sequencer	all nodes	Broadcast message
RETRANS	node	sequencer	Request asking for a missed message
PHASE1	sequencer	all nodes	Request to start two-phase commit protocol
PHASE2	sequencer	all nodes	Request to start second phase
ACK_COMMIT	node	sequencer	Inform that a phase has been completed

The *SequenceNr* field is used by the sequencer to sequence broadcast messages and by users to acknowledge the last broadcast received so far. The *MessageNr* field holds the sender's *MessageId* value, and thus identifies the message. It is used for detecting duplicate messages. The *SenderID* and *DestID* are used to identify the sender and destination, respectively.

When an application process wants to do a broadcast, it invokes its local kernel and passes the data to be broadcast. The kernel then executes the algorithm given in Fig. 3.

```
broadcast(data)
char *data;
{
/* Algorithm executed by the kernel on the machine wanting to broadcast. */
    struct header h;                        /* outgoing header is built here */

    h.type = DATA;                          /* DATA means request for broadcast */
    h.SequenceNr = LastSeqReceived;         /* piggybacked acknowledgement */
    h.MessageNr = MessageId;                /* unique message identifier */

    send(sequencer, &h, data);              /* point-to-point message to sequencer */
    MessageId++;                            /* use different message id next time */
    StartTimer();                           /* just in case broadcast does not arrive */
    block();                                /* wait for broadcast */
}
```

Fig. 3. Algorithm used by sending kernel to achieve reliable broadcast.

Having looked at what the sender does to transmit a message to the sequencer for broadcast, let us now turn to the sequencer to see what it does with the message when it comes in (see Fig. 4). The sequencer first checks if the message is a duplicate, using the message number in the header field *MessageNr*. If so, it informs the sender that the message already has been broadcast.

If the message is new, the sequencer calls the function *FullHistory*, which tries to remove messages from the history using the piggybacked acknowledgements stored in *SequenceNr* table when the history is full. If there is room in the history, the sequencer assigns the message the sequence number *NextSeqToUse*, stores the message in the history, and broadcasts the message. It also increments *NextSeqToUse* for next time.

If the history buffer is still full the sequencer enters the synchronization phase to bring all nodes up-to-date and to empty its history, as will be discussed below.

```
DataMessage(h, data)
struct header *h;
char *data;
{
/* Algorithm used by the sequencer when a request-to-broadcast arrives. */

    struct message *m, *GetFromHistory();

    if (h–>MessageNr == MessageNr[h–>SenderID]) {
        /* This is a duplicate. Sender must have timed out. */
        m = GetFromHistory(h–>MessageNr, h–>SenderID);    /* Fetch sequence number from buffer */
        send(h–>SenderID, m–>header, NULLDATA);           /* Tell sender */
    } else {
        /* This is a new message that has not yet been broadcast. */
        SequenceNr[h–>SenderID] = h–>SequenceNr;          /* Accept piggybacked ack. */
        if (!FullHistory()) {                             /* is history full? */
            /* No, there is room. */
            MessageNr[h–>SenderID] = h–>MessageNr;        /* Save unique message id. */
            h–>SequenceNr = NextSeqToUse;                 /* assign sequence number. */
            StoreInHistory(h, data);                      /* save message */
            PassToApplication(data);                      /* Upcall to application to process data */
            NextSeqToUse++;                               /* for next time */

            h–>type = BROADCAST;
            broadcast(h, data);                           /* do the broadcast */
        } else
            EnterSyncPhase();                             /* history is full, flush it. */
    }
}
```

Fig. 4. Code executed by the sequencer when it receives a request-to-broadcast.

The protocol requires three algorithms to be executed. First, the sender must build a message and transmit it to the sequencer. Second, the sequencer must process the incoming message and broadcast it. Third and last, the user nodes must handle arriving broadcast messages. We have already described the first two steps; now let us look at the last one.

When a broadcast message arrives, the receiving kernel executes the procedure *AcceptBroadcast* (see Fig. 5). It first checks to see if the sequence number is the one it is expecting. If so, it increases *LastSeqReceived*. If the sequence number is not the expected one, the node has missed one or more broadcasts and asks the sequencer for a retransmission of the missing message(s). Out-of-sequence broadcast messages may be buffered in the kernel, but the kernel is required to pass messages to the application in the correct order. If the kernel does not receive a missing message within a certain number of retries, it assumes that the sequencer has failed and a subprotocol to elect a new sequencer is started.

```
AcceptBroadcast(h, data)
struct header *h;
char *data;
{
    struct header rh;

/* Algorithm executed by the kernel when a broadcast message arrives. */
    if (h–>SequenceNr == LastSeqReceived) {
        /* This broadcast message contains the sequence number expected. */
        LastSeqReceived++;                          /* one more broadcast received */
        PassToApplication(data);                    /* pass to application */
    } else {
        /* Wrong sequence number. We have missed some broadcasts. */
        rh.type = RETRANSMIT;                       /* ask sequencer for the missing ones */
        rh.SequenceNr = LastSeqReceived;            /* tell where we are */
        send(SequencerID, &rh, NULLDATA);           /* send to sequencer */
        SetTimer();                                 /* make sure that we get a reply */
    }
}
```

Fig. 5. Algorithm for processing an incoming broadcast.

If a node misses a broadcast from the sequencer, this failure will eventually be detected while handling subsequent messages. The assumption is that there will be subsequent messages. This assumption need not be true, depending on the communication patterns of the applications. For example, a node may send a message that triggers another process to send messages. If this process misses the message, the system may very well become deadlocked. To prevent this from happening, each node sends a dummy message after receiving a certain number of messages without sending any. This is effectively a primitive timer; a real one could also be used, but this would be less efficient. This dummy message contains *LastSeqReceived* to inform the sequencer which messages it has received.

2.2. Synchronization phase

The sequencer keeps a fixed-size history of messages it has broadcast. It may happen that the buffer gets full, because it has not received any messages from one of the nodes for a period of time and the dummy messages sent by the quiet node have gotten lost. If, despite all precautions, this happens, the sequencer enters the synchronization phase to empty its history. The synchronization phase runs a two-phase commit protocol to guarantee that all participants received all messages [Eswaran et al. 1976]. During the synchronization phase, the sequencer does not accept any data messages and all nodes stop sending data messages.

In the first phase, the sequencer broadcasts an intention message (PHASE1) to all nodes, containing *NextSeqToUse* – 1. This is the number of the last message broadcast. When a node receives an intention message, it enters the synchronization phase and checks if it has missed any messages, using the sequence number stored in the intention message. If it has missed anything, it sends a request to the sequencer to retransmit the missing message(s). If, on the other hand, the node is up-to-date, it informs the sequencer. Either way, it sends the sequencer a message acknowledging the broadcasts it has received.

The messages from all the nodes are used to update the sequencer's *SequenceNr* table. In this way the sequencer is brought up to date on the status of all the nodes, and all the nodes receive all messages that have been broadcast. The sequencer goes to the second phase as soon as it has received a reply from all nodes that all messages have been received.

In the second phase, the sequencer empties its history and informs all other nodes that every node is up-to-date. When a node receives a commit message (PHASE2), it sends an ACK-COMMIT, and goes back to normal operation. When the sequencer has received an ACK-COMMIT from all nodes, it leaves the synchronization phase and goes back to normal operation.

3. IMPLEMENTATION

We have built a prototype kernel running the protocol described above. The prototype runs on the bare hardware, rather than on top of an operating system. It has been used for running parallel applications written in the programming language Orca [Bal and Tanenbaum 1988; Bal et al. 1989a; Bal et al. 1989b]. It uses the Amoeba protocols [Van Renesse et al. 1989] to communicate with our local UNIX and Amoeba systems.

The prototype runs on two different systems. One system is a multiprocessor with 10 16 Mhz MC68020 CPUs. The system contains 8Mb of shared memory, which is accessible through a VME bus. This implementation uses the shared memory to simulate unreliable messages. The reliability of the network (i.e., the percentage of broadcast messages delivered at a destination) is an adjustable parameter of the system. In this way, we are able to test our protocol with different degrees of reliability. The second implementation runs on a real distributed system, containing 10 16 Mhz MC68020 CPUs connected to each other through an 10 Mbit/s Ethernet [Metcalfe and Boggs 1976]. This implementation uses Ethernet multicast communication to broadcast a message to a group of processors. All processors are on one Ethernet and are connected to the network by Lance chip interfaces (manufactured by Advanced Micro Devices). We will give performance numbers for this implementation.

We have done several experiments to measure the performance of the protocol. In the first experiment, one node continuously broadcasts null-messages to a group of nodes. This experiment measures the delay seen from the sending user process, between calling and returning from the broadcast primitive. The sending process runs on a different processor than the sequencer. Note that this is the worst possible case for our protocol, since only one 1 processor sends messages to the sequencer (i.e., no acknowledgements can be biggybacked by other processors). The second property that the first experiment tests is the scalability of the protocol.

The results of the first experiment are depicted in Fig. 6. For a group of two nodes, the measured delay is 1.3 msec. Compared to the Amoeba RPC [Van Renesse et al. 1989], which claims to be the world's fastest distributed operating system, our broadcast message costs 0.1 msec less than Amoeba's RPC. Amoeba RPC uses three packets, whereas the broadcast protocol uses on average 2.5 packets in this experiment (2 for the broadcast and once in a while a packet from every node to tell the sequencer that it is up-to-date). For a group of 10 nodes, the measured delay is 1.5 msec. Although the protocol does not scale

Group size	2	3	4	5	6	7	8	9	10
Delay (msec)	1.32	1.35	1.38	1.40	1.43	1.45	1.48	1.50	1.53

Fig. 6. Performance per message for one site broadcasting continuously. *HISTORY_SIZE* is 20.

perfectly, a reliable broadcast to a group of 10 processors is still less expensive than most RPC implementations. From the other numbers in the table, one can conclude that each node adds 25 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 3.8 msec.

We have also measured the performance of the protocol on a heavily-loaded system. In this experiment, a number of senders broadcast null-messages to a group of 10 nodes. This experiment measures how the performance of the protocol degrades with the number of sending processes. This should be no worse than just dividing the maximum performance over the senders. That is, if one sender can do 667 broadcasts per second, then two senders together should be able to do a total of at least 667 broadcasts per second as well. We also want to know how fairly the broadcasts are distributed over the senders. If, with two senders, one could execute only 5 broadcasts and the other sender 662 the sequencer would be unfair.

# Senders	2	3	4	5	6	7
Broadcast/sec	1333	1428	1538	1562	1500	1521
Delay (msec)	1.5 ± 0.0	2.1 ± 0.0	2.6 ± 0.01	3.2 ± 0.01	4.0 ± 0.02	4.6 ± 0.02

Fig. 7. Performance under heavy load for a group of 10 nodes. *HISTORY_SIZE* is 20.

To measure the performance and the fairness of the protocol, we have measured the total number of broadcasts, the average delay for one broadcast, and the deviation from the average delay for one broadcast, as a function of the number of senders. In this experiment, one of the senders is running on the same node as the sequencer. As can been seen in Fig. 7, the number of broadcast per second for a number of senders is even better than for one sender. For a large number of senders the performance decreases slightly due to the large number of collisions. The second number in the second row gives the average deviation from the delay. The fairness is almost ideal.

Our measurements put an extreme load on the network. Measurements under these conditions for larger messages make no sense, because the hardware cannot keep up. For example, three senders that broadcast 1000 bytes messages use almost 80% of the bandwidth of the Ethernet. In practice, however, our applications that are based on the broadcast protocol have used at most 2% of the capacity of the Ethernet.

4. DISCUSSION AND COMPARISON

A number of broadcast protocols have been published in the literature. Most of them are not concerned with a broadcast medium, but with a network with point-to-point communication links. We will compare our protocol with two well-known protocols for reliable broadcast and with three that have been published recently. For other papers on broadcast protocols we refer the reader to the list of references [Amano 1987; Ahamad and Bernstein 1985; Burr 1984; Crowcroft and Paliwoda 1988; Even and Awerbuch 1984; Garcia-Molina and Kogan 1988; Ho and Johnson 1986; Mockapetris 1983; Plata and Zapata 1986; Schneider et al. 1984; Segall and Awerbuch 1983; Topkis 1985; Wong and Gopal 1983].

Chang and Maxemchuk (CM) describe a family of protocols [Chang and Maxemchuk 1984]. The protocols differ mainly in the degree of fault-tolerance that they provide. Our protocol as described in this paper resembles the protocol that is not fault-tolerant (i.e., it cannot recover from processor crashes), but ours is optimized for the common case of no communication failures. Like our protocol, the CM protocol depends also on a central node, the token site, for ordering messages. However, on each acknowledgement another node takes over the role of token site. Depending on the system utilization, the transfer of the token site on each acknowledgement can cost one extra control message. Thus their protocol requires 2 to 3 messages per broadcast, whereas ours requires only 2 in the normal case. In addition, the CM protocol uses more memory space, because all data messages are saved on all nodes until the acknowledgement arrives. 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 determined by the transmission time, but mainly by the processing time at the nodes. In their scheme, each broadcast causes at least $2(n-1)$ interrupts, ours only n.

Birman and Joseph (BJ) describe a protocol that uses a distributed two-phase protocol to order all messages [Birman and Joseph 1987]. As this protocol uses a large number of messages, they propose other protocols that relax the ordering semantics. Our protocols could also be used to relax the ordering semantics by using, for example, one sequencer per broadcast group. The protocols are hard to compare, as the BJ protocols are concerned with achieving fault-tolerance.

Navaratnam, Chanson, and Neufeld (NCN) have described a protocol very similar to the CM protocols [Navaratnam et al. 1988]. The NCN protocol uses also a centralized scheme, but instead of transferring the token site on each acknowledgement, their central site waits until it has received acknowledgements from all receivers 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. Ours is thus an order of magnitude faster.

A totally different approach to reliable broadcast is described by Melliar-Smith and Moser (MM) [Melliar-Smith and Moser 1989]. They describe a protocol that achieves reliable broadcast with a certain probability. They claim that the probability is high enough to assume that all messages are ordered globally, but nevertheless there is a certain chance that messages are not globally ordered. Although the MM protocol is uses few messages, messages cannot be delivered to 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.

The last different approach we discuss is by Garcia-Molina and Spauster [Garcia-Molina and Spauster 1989], which is mainly concerned with the problem of overlapping groups. For a given set of multicast groups, a graph is generated, called the propagation graph. The graph indicates the paths messages should follow to get to all destinations. While messages propagate along these paths, they are ordered globally by merging messages destined for different groups. Although this method cannot fully profit from a broadcast medium, it costs one broadcast message for all destinations in the group and a number of messages to propagate along the paths. In addition, a message can only be delivered to an application after is has traversed the paths in the propagation graph.

It is hard to compare precisely the described protocols with ours, because most protocols are tailored for fault-tolerance. If one omits the reliability of the network, three factors determine the efficiency of the described protocols. One is the number of messages, second is the delay before the message can be delivered to the application, and third is the number of interrupts. In our protocol, the number of messages used is determined by the size of the history and the communication pattern of the application. In the normal case, 2 messages, a point-to-point message to the sequencer and a broadcast message, are used. In the worst case, when one node is continuously broadcasting, (*#nodes/HISTORY_SIZE*) + 2 are needed. For example, if the number of buffers in the history is equal to the number of processors, 3 messages per reliable broadcast are needed. In practice, with say 1Mb history buffers and 1Kb messages, there is room for 1024 messages. This means that the history buffer will rarely fill up and the protocol will actually average 2 messages per reliable broadcast.

The delay before a message can be delivered to the application is optimal; as soon as a broadcast arrives, it can be delivered. Also, our protocol causes a low number of interrupts. Each node gets one interrupt for each reliable broadcast message. The storage requirement for our protocol at the sequencer node is equal to the number of buffers needed for the history. All other nodes, however, do not have any special storage requirements. If one ignores messages and delays that are needed to achieve fault-tolerance, our protocol is more efficient than any of the described protocols.

5. CONCLUSIONS

We have presented a simple protocol that achieves reliable broadcast and guarantees that all messages will be received by every live node in the same order. In addition, we have implemented our protocol and given performance measurements for a reliable broadcast on a 10 Mbit/s Ethernet with 16 Mhz MC68020 CPUs. The protocol performs a reliable broadcast to 10 nodes in 1.5 msec.

The protocol uses a centralized node (the sequencer) to determine the order of the messages. Although this centralized node does not do anything computationally intensive (it receives a message, adds the sequencer number, and broadcasts it), it could become a bottleneck in the system. In practice, however, this has never happened. The most communication intensive application that we have run performed 27 broadcasts/sec. This is less than 2% of the maximum broadcast/sec that the protocol can generate before the network interfaces saturate. Nevertheless, the sequencer could become a bottleneck and prevent the system from scaling to a very large number of nodes. We therefore are looking into special hardware support that would reduce the possibility of such a bottleneck.

If the sequencer fails, the whole system will come to a grinding halt. However, it is easy to extend our protocol to recover from sequencer failures. One idea is to replicate the history and to elect the node with the highest sequence number as the new sequencer. When the histories are replicated, the newly elected sequencer can bring the other nodes up-to-date if they have missed messages. The only messages that are completely lost are those that were sent by an application running on the same processor as the sequencer kernel and that were missed by all processors. As long as at least one of the surviving processor receives each broadcast, the entire history can be reconstructed.

6. ACKNOWLEDGEMENTS

We would like to thank Hans van Staveren for his help with implementing the protocol and Erik Baalbergen, Mathilde van Es, and Robbert van Renesse for their critical reading and useful comments.

7. REFERENCES

Ahamad, M. and Bernstein, A.J., ‘‘Multicast Communication in Unix 4.2BSD,’’ *Proc. 5th Int. Conf. on Distr. Comp. Syst.*, pp. 80-87, Denver, CO (May 1985).

Amano, H., ‘‘RSM (Receiver Selectable Multicast),’’ *Proc. Int. Conf. on Computers and Applications*, pp. 149-156, Peking, China (June 1987).

Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., ‘‘A Distributed Implementation of the Shared Data-Object Model,’’ *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL (Oct. 1989a).

Bal, H.E., Renesse, R. van, and Tanenbaum, A.S, ‘‘Implementing Distributed Algorithm using Remote Procedure Call,’’ *Proc. National Computer Conference, AFIPS*, pp. 499-505, Chigaco, IL. (1987).

Bal, H.E., Steiner, J.G., and Tanenbaum, A.S., ‘‘Programming Languages for Distributed Computing Systems,’’ *ACM Comp. Surv.*, Vol. 21, No. 3 (Sept. 1989b).

Bal, H.E. and Tanenbaum, A.S., ‘‘Distributed Programming with Shared Data,’’ *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, pp. 82-91, Miami, FL (Oct. 1988).

Birman, K.P. and Joseph, T.A., ‘‘Reliable Communication in the Presence of Failures,’’ *ACM Trans. on Comp. Syst.*, Vol. 5, No. 1, pp. 47-76 (Feb. 1987).

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

Burr, W.E., ‘‘A Fault-Tolerant Hierarchical Broadcast Network,’’ *Proc. Computer Networking Symp.*, pp. 11-17, Garthersburg, MD (Dec. 1984).

Chang, J. and Maxemchuk, N.F., ‘‘Reliable Broadcast Protocols,’’ *ACM Trans. on Comp. Syst.*, Vol. 2, No. 3, pp. 251-273 (Aug. 1984).

Crowcroft, J. and Paliwoda, K., ‘‘A Multicast Transport Protocol,’’ *ACM Comput. Commun. Rev.*, Vol. 18, No. 4, pp. 247-256 (1988).

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., ‘‘The Notions of Consistency and Predicate Locks in a Database System,’’ *Commun. ACM*, Vol. 19, pp. 626-633 (Nov. 1976).

Even, S. and Awerbuch, B., ‘‘Efficient and Reliable Broadcast is Achievable in a Eventually Connected Network,’’ *Proc. of the 3rd Annual Symp. on Princ. of Distr. Comp.*, Miami/Ft. Lauderdale, FL (Aug. 1984).

Garcia-Molina, H. and Kogan, B., ‘‘Reliable Broadcast In Networks with Nonprogrammable Servers,’’ *Proc. 8th Int. Conf. on Distr. Comp. Syst.*, pp. 428-437, San Jose, CA (June 1988).

Garcia-Molina, H. and Spauster, A., ‘‘Message Ordering in a Multicast Environment,’’ *Proc. 9th Int. Conf. on Distr. Comp. Syst.*, pp. 354-361, Newport Beach, CA (June 1989).

Ho, C-T. and Johnson, S.L., ‘‘Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes,’’ *Proc. of the 1986 Int. Conf. on Parallel Processing*, pp. 640-648, St. Charles, IL. (Aug. 1986).

Lawler, E.L. and Wood, D.E., ‘‘Branch and Bound Methods: A Survey,’’ *Operations Research*, Vol. 14, pp. 699-719 (July 1966).

Melliar-Smith, P.M. and Moser, L.E., ‘‘Fault-Tolerant Distributed Systems Based on Broadcast Communication,’’ *Proc. 9th Int. Conf. on Distr. Comp. Syst.*, pp. 129-134, Newport Beach, CA (June 1989).

Metcalfe, R.M. and Boggs, D.R., ‘‘Ethernet: Distributed Packet Switching for Local Computer Networks,’’ *Commun. ACM*, Vol. 19, No. 7, pp. 395-404 (July 1976).

Mockapetris, P.V., ‘‘Analysis of Reliable Multicast Algorithms for Local Networks,’’ *Proc. 8th Data Commun. Symp.*, pp. 150-157, Silver Spring, MD (Oct. 1983).

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

Plata, O. and Zapata, E.L., ‘‘Optimal Broadcasting Figure in Computer Networks: An Algorithmic Solution,’’ *Proc. Computer Network Symp.*, Washington, DC. (Nov. 1986).

Renesse, R. van, Staveren, J.M. van, and Tanenbaum, A.S., ‘‘The Performance of the Amoeba Distributed Operating System,’’ *Software—Practice and Experience*, Vol. 19, No. 3, pp. 223-234 (March 1989).

Schneider, F.B., Gries, D., and Schlichting, R.D., ‘‘Fault-Tolerant Broadcasts,’’ *Science of Computer Programming*, Vol. 4, pp. 1-15 (1984).

Segall, A. and Awerbuch, B., ‘‘A Reliable Broadcast Protocol,’’ *IEEE Trans. on Comm.*, Vol. 31, No. 7, pp. 896-901 (July 1983).

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

Topkis, D.M., ‘‘Concurrent Broadcast for Information Dissemination,’’ *IEEE Trans. on Soft. Eng.*, Vol. 11, No. 10, pp. 1107-1112 (Oct. 1985).

Wong, J.W. and Gopal, G., ‘‘Analysis of Reliable Broadcast in Local-Area Networks,’’ *Proc. 8th Data Commun. Symp.*, pp. 158-163, Silver Spring, MD. (Oct. 1983).