An Evaluation of the Amoeba Group Communication System

M. Frans Kaashoek
Laboratory for Computer Science
M.I.T.
Cambridge, U.S.A.

Andrew S. Tanenbaum
Dept. of Math and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Abstract

The Amoeba group communication system has two unique aspects: (1) it uses a sequencer-based protocol with negative acknowledgements for achieving a total order on all group messages; and (2) users choose the degree of fault tolerance they desire. This paper reports on our design decisions in retrospect, the performance of the Amoeba group system, and our experiences using the system. We conclude that sequencer-based group protocols achieve high performance (comparable to Amoeba's fast remote procedure call implementation), that the scalability of our sequencer-based protocols is limited by message processing time, and that the flexibility and modularity of user-level implementations of protocols is likely to outweigh the potential performance loss.

1 Introduction

Group communication allows applications to send data to n destinations using a single message. Applications using group communication instead of point-to-point communication are potentially easier to write and perform better. This paper reports on our design decisions in retrospect, the performance of the Amoeba group communication system, and our experiences using the system.

The semantics of the Amoeba group primitives are simple, powerful, and easy to understand. The primitives, for example, guarantee total ordering of group messages. The proposed primitives are also efficient: if a network supports physical multicast, a reliable group send can be done in just slightly more than two messages on the average, so that the performance of a reliable group send is roughly comparable to that of a remote procedure call (RPC) [6]. In addition, the primitives are flexible: user applications can, for example, trade performance against fault tolerance.

The Amoeba group communication primitives have been implemented in the kernel of the Amoeba distributed operating system [21, 31]. 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 second per group. With multiple groups, the maximum number of broadcasts per second has been measured at 3175. In addition, we learned that (1) the scalability of our sequencer-based protocols is limited by message processing time; and (2) the flexibility and modularity of user-level implementations of protocols is likely to outweight the potential performance loss.

The rest of this paper is structures as follows. Section 2 describes the main design decisions. Section 3 briefly overviews the implementation. Section 4 provides a detailed performance analysis of the group system. Section 5 discusses the design decisions and our experience with implementing and using the system. Section 6 relates the Amoeba group system to other systems. Section 7 summarizes our main conclusions.

2 The Amoeba Group System

A group is a collection of processes that can communicate directly with each other using 1-to-many messages. Table 1 summarizes the primitives offered by Amoeba for group communication. To send to or to receive from a particular group, a process has to join the group. 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.

We have chosen to make the primitives blocking to sim-

This research was performed at the Vrije Universiteit as part of the first author's Ph.D. thesis

Primitive	Description
CreateGroup	Create a group.
JoinGroup	Join a group.
SendToGroup	Send a message to a group.
ReceiveFromGroup	Receive a message from a group.
ResetGroup	Reform group after a processor failure.
GetInfoGroup	Return state information about a group.
ForwardRequest	Forward an RPC request to another group member.

Table 1. Group communication interface.

plify programming. Parallelism can be obtained by multithreading the application. This decision is consistent with our RPC interface for point-to-point communication.

Two aspects are unique to the Amoeba group communication system: (1) Amoeba user's can select the degree of fault tolerance on group messages; (2) Amoeba uses an efficient sequencer-based protocol with an negative-acknowledgement scheme to achieve total ordering. We discuss the reasons that led to these design decisions in turn.

2.1 Reliability

Amoeba's group primitives offer reliable communication: the group protocol automatically recovers from lost, garbled, and duplicate messages. Although Amoeba's network protocol supports unreliable group communication [19], we decided to make only reliable group communication available to the programmer. This decision 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.

At 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 [10]. To be able to reach a decision about whether a process is alive, the algorithm sends messages asking the recipient to respond. If after a certain number of trials a process does not respond, the process is declared "dead." Using this unreliable failure-detection method some processes may be declared dead although they are functioning fine (e.g., when a process does not respond fast enough). Dead processes are removed from the group so that they cannot cause any problems for the remaining living processes.

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 crash failures [27]. 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 broadcast semantics, failures, protocols for different types of failures, see Hadzilacos and Toueg [11].

2.2 Ordering

The group primitives guarantee a total ordering (with FIFO) per group. If two members send messages A and B concurrently, the protocol guarantees that all members of the group either receive first message A and then B, or first B and then A. It never happens that member 1 sees A and then B, and member 2 sees B and then A. Many distributed applications are easy to implement with a total ordering, as the programmer can think of processes running in lockstep [28].

In the past, most designers have chosen weaker protocols than we have, making application building more difficult. 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 [7].

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 more complex, and often perform worse. For example, the distributed protocols for total ordering in Isis have been replaced by a dynamic-centralized protocol because the distributed one was too complex and slow [5]. 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 enormous 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. From experience we have observed that for many applications, hundreds of messages per second is sufficient.

To keep the protocol simple we decided to use a *static* sequencer-based protocol. This decision simplifies the implementation because the sender of message always knows where the sequencer is, but may result in a loss of performance, as a message has to be transmitted first to the sequencer. After publication of our initial results showing that a sequencer-based protocol performs well [17], a number of designers of group systems have chosen to use a sequencer-based protocol; most newer systems, such as Horus [33] and Transis [1], use a *dynamic* sequencer-based protocol, in which the sequencer migrates to the sender so that the next message can be sent without having to go remotely to acquire a sequence number. Initial experience with these systems indicates that this is an appropriate choice when messages come in bursts.

There are two reasons for using a negative acknowledgement scheme. First, it reduces the number of acknowledgement messages. In a *positive acknowledgement scheme*, a process sends an acknowledgement back to the sender for a message. This works fine for point-to-point messages, but less well for broadcast messages. If a process sends a broadcast message to a group, with say 256 members, 255 acknowledgements will be sent back to 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 sending a message for when a process has *not* received a message 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. This approach is attractive in unreliable networks, but it causes far more acknowledgements to be sent than with a negative acknowledgement scheme; it just spreads the acknowledgement load out over time.

3 Implementation

This section gives a brief summary of the implementation of the Amoeba group system. A detailed description of the implementation can be found elsewhere [14, 16]. We give enough information to be able to understand the performance experiments described in the next section.

3.1 The broadcast protocols

In the common case, the Amoeba broadcast protocol uses two messages per *SendToGroup*. One point-to-point message from the sending process to the sequencer and one multicast message stamped with a new sequence number from the sequencer to the group. We call this method *PB method*.

There is another two-messages protocol possible, which we call the *BB method*. In the BB method, the sender multicasts the message. When the sequencer sees this multicast message, it multicasts a special *accept* message containing the newly assigned sequence number. A multicast message is only "official" when the *accept* message has been sent.

The PB method uses bandwidth to reduce the number of interrupts, while the BB methods minimizes bandwidth usage at the cost of more interrupts. The PB method sends each message twice on the network, consuming $2n$ bytes of network bandwidth, where n is the number of bytes of user data. The BB method sends the full message only once over the networking, consuming n bytes of network bandwidth. On the other hand, every machine is interrupted twice, once for the message and once for the short *accept* message. The implementation switches dynamically between the PB and BB methods depending on message size.

To recover from lost and garbled messages the sequencer keeps a history buffer of all recently sent messages. The sequencer deletes a message from the history buffer when it knows all processes in the group have received the message. In order to do so, each process piggybacks on each message

Group Communication	RPC
FLIP Layer	
Network with multicast	Network without multicast

Table 2. Communication layers in the Amoeba kernel.

it sends to the sequencer the sequence number of the last message it has received.

If users desire recovery from group member failures, they can specify a *resilience degree* when calling *CreateGroup*. A resilience of degree r means that the *SendtoGroup* primitive does not return control to the application until its 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. (For clarity we use the PB method, but the protocol works for the BB method too.) The sequencer allocates the next sequence number, but does not officially accept the message yet. Instead, it buffers the message and broadcasts to the group the message and sequence number as a *tentative* broadcast to the group. When a member kernel receives this tentative broadcast, it buffers the message in its history buffer 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 r 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.

3.2 Implementation structure

The group protocols and the underlying routing protocols are implemented in the Amoeba kernel. The communication system in the kernel consists of 3 layers (see Table 2). The top layer implements the protocols for group communication and RPC. The protocols described in the previous section are implemented here.

Both the RPC and group communication modules use the Fast Local Internet Protocol (FLIP) [14] to send messages. FLIP is a connectionless (datagram) protocol, roughly analogous to IP [25], but with increased functionality. For the experiments performed in this section we could have used multicast-IP [9] 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.

4 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 traffic is within a single network 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 bytes. The last size was chosen to reflect a limitation in our implementation. In principle, the group communication protocols can handle messages larger than 8,000 bytes, but lower layers in the kernel make it impossible to measure the communication costs for these sizes in a meaningful way. 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 [29], but it is not immediately clear how these should be extended to multicast communication. Some recent progress has been made in this area [1], 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 an arbitrary, but reasonable 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 bytes, 1 Kbyte, 4 Kbyte, and 8,000 bytes to a group of processes (the size of

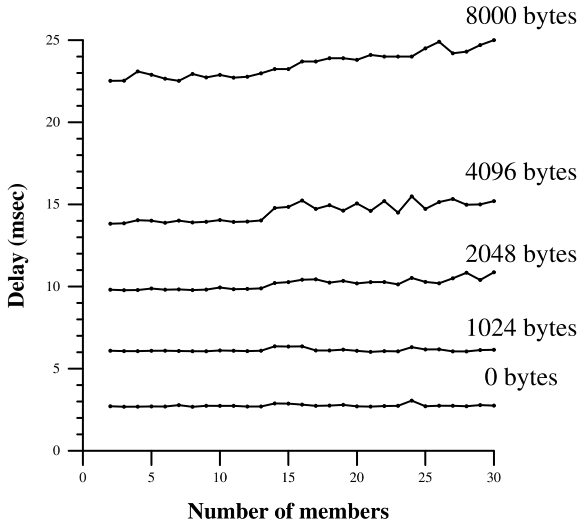

Figure 1. Delay for 1 sender using PB method ($r = 0$).

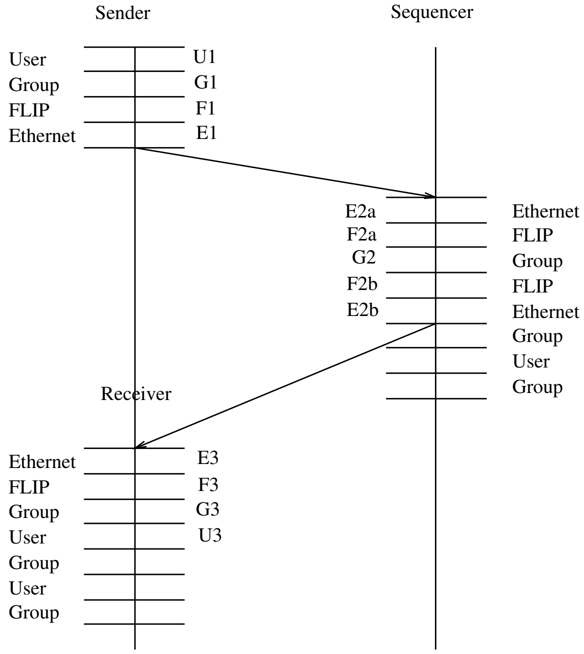

Figure 2. A break down of the events in a single *SendToGroup* - *ReceiveFromGroup* pair. The group size is 2 and the PB method is used

the message excludes the 116[1] 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 1. 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 0.1 msec faster 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 microseconds 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 2 and Table 3 break 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 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 microseconds.

The results of the same experiment but now using the BB method are depicted in Figure 3. 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

Layer	Parts	Microseconds
User	U1+U2	514
Group	G1+G2+G3	740
FLIP	F1+F2a+F2b+F3	570
Ethernet	E1+E2a+E2b+E3	916
Total		2740

Table 3. The time spent in the critical path of each layer. The Ethernet time is the time spend on the wire plus the time spend in the driver and taking the interrupt.

[1] 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.

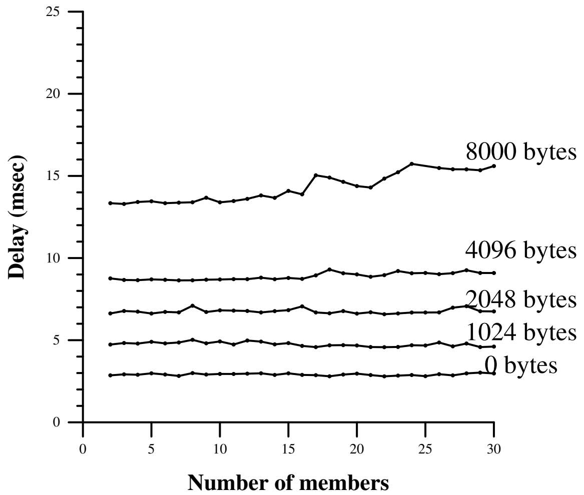

Figure 3. Delay for 1 sender using BB method ($r = 0$).

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 4 and Figure 5. 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 microseconds, 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 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 send directly from main memory, this last copy could have been avoided.) If Amoeba had support for sophisticated memory management primitives like Mach [36], 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, although manipulating the memory maps also

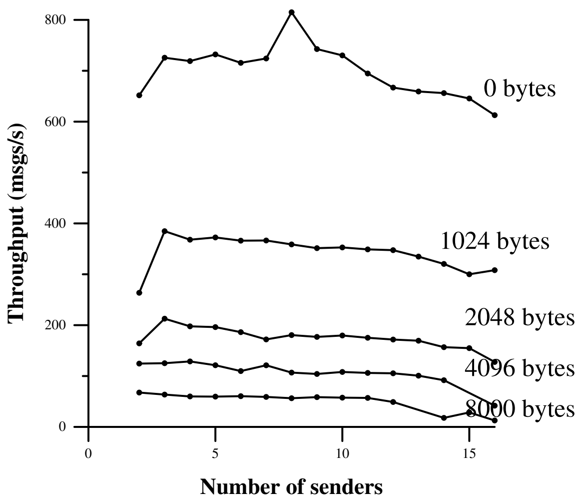

Figure 4. Throughput for the PB Method. The group size is equal to the number of senders.

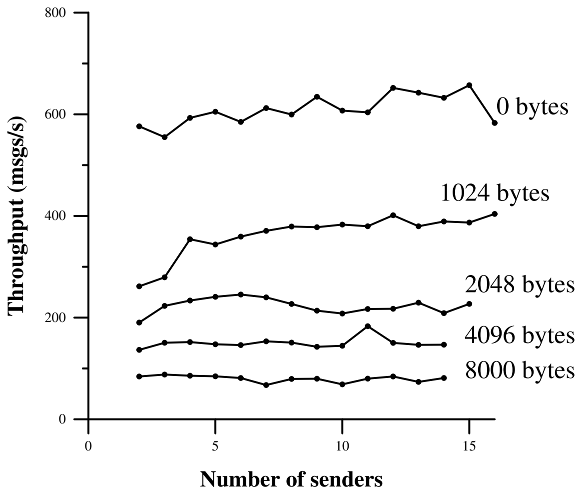

Figure 5. Throughput for the BB Method. The group size is equal to the number of senders.

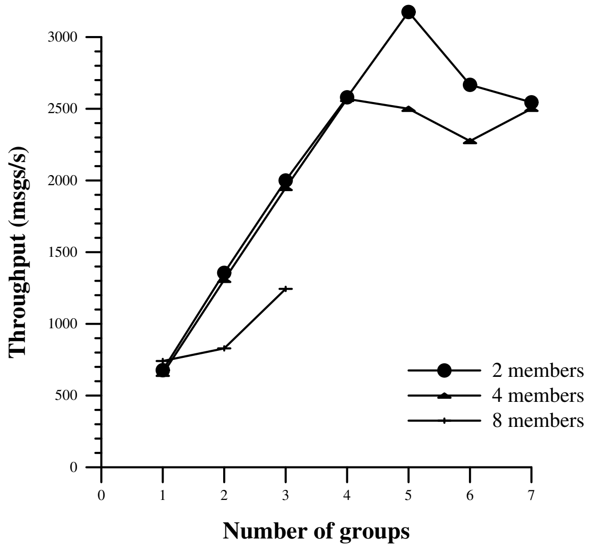

Figure 6. Throughput for groups of 2, 4, and 8 members running in parallel and using the PB method. We did not have enough machines available to measure the throughput with more groups with 8 members.

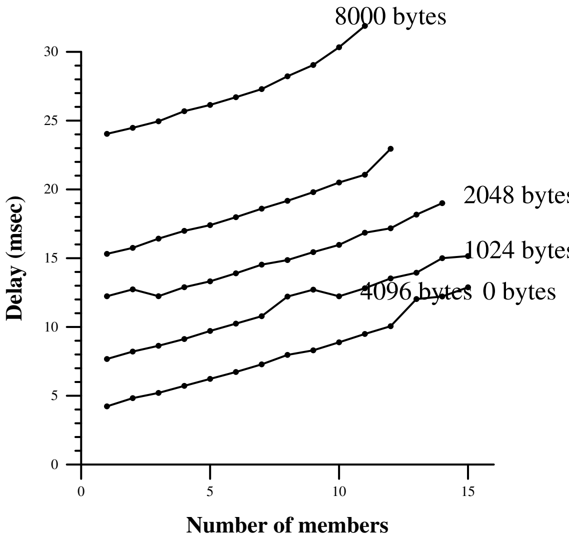

Figure 7. Delay for 1 sender with different rs using the PB method. Group size is equal to $r + 1$.

takes some time.

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 problem arises because 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 processor. The results are depicted in Figure 6. 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 cause for the poor performance of groups of size 8. Note that the Ethernet utilization at this data rate is 61%, which is as much as can be expected from an Ethernet with multiple uncoordinated senders. With a faster network, performance will be higher.

The final experiment measures the delay of sending a message with $r > 0$. Figure 7 and Figure 8 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 microseconds.

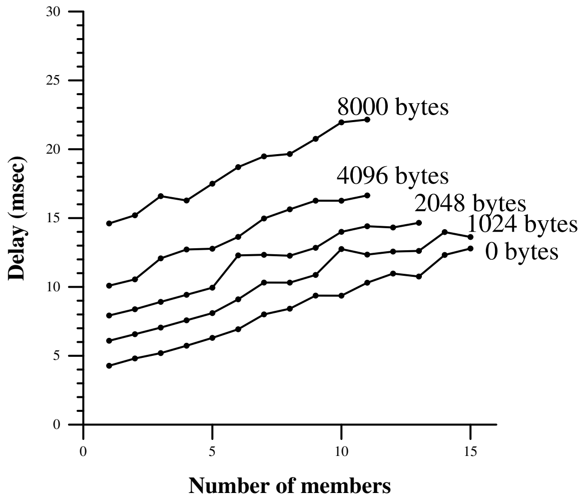

Figure 8. Throughput for the PB Method. The group size is equal to the number of senders.

5 Discussion

In December 1991 the Amoeba 5.0 kernel, which contains the group protocols presented in the paper, replaced the previous generations of Amoeba kernels and has been in day-to-day use at the Vrije Universiteit since then. Currently over 100 computers with different CPUs (Intel 386/486, SPARCs, and MC680x0) are running this kernel. Since late 1992 this version of the Amoeba system has been publicly and commercially available; over 150 sites have picked the system up. In 1993 the core of the group communication protocols was incorporated in the Panda system [2], a portable platform for parallel computing, which runs on cluster of UNIX workstations and supercomputers, such as a 512-node Parsytec and a 128-node CM-5 [12, 13]. Both the supercomputers provide reliable communication, so the protocols on these machines are less complex. The group communication primitives have been used in running parallel applications [15, 18, 30, 35]. This section reviews some of the design decision given our experience with using and implementing the group communication protocols. We will focus on the lessons learned.

The sequencer-based protocol has proven to be an efficient, simple, and robust protocol to implement and to use. Except for synthetic benchmarks, the limit imposed by the sequencer on the maximum throughput has seldom been a problem in applications. In most applications the delay of sending a message is more important. In applications that have performed badly, the performance was not limited by the number of messages the sequencer could process, but by the time that the individual processors needed to process each message. Therefore, we have concluded that it is more important to reduce the software overhead of message processing than to make the protocol more distributed. We are currently experimenting with a new approach, called optimistic active messages, which reduces this software overhead for message processing [34].

In some applications one process sends multiple messages before the next process sends a message. The performance of these applications could have benefited from a migrating sequencer, as used in more recent systems such as Horus [33] and Transis [1]. Instead, we found ourselves placing the process that is sending most messages on the kernel that runs the sequencer. In retrospect, the performance gained by migrating the sequencer may be worth the additional complexity in the protocol for distributing the history buffer.

Amoeba applications using group communication fall into two broad categories: (1) parallel computations and (2) replicated servers. Although we have developed and implemented a consistent checkpointing scheme for parallel applications [15], most of the parallel applications are just restarted if a processor failures happens. All of them run with a resilience degree of zero. The replicated servers tend to run in small groups (about 3 members) and the overhead for the acknowledgements for a higher resilience degree is acceptable. Making the resilience degree a user-settable parameter has allowed the group communication protocols to be used both for parallel and fault-tolerant applications and has made the performance of the group system comparable to the RPC system that Amoeba supports for point-to-point communication.

The support for applications that need to be fault-tolerant was initially inadequate. We expected that building fault-tolerant programs with the group primitives was going to be relatively straightforward. However, we underestimated two important aspects of building a fault-tolerant applications. First, the system did not provide any support for the atomic creation of a group. In a system with unreliable communication and failures, atomic group creation is theoretically impossible to achieve, but a heuristic library procedure that does an "best efforts" attempt as good as possible would have simplified building some of the early fault-tolerant programs. Second, the system did not have good support for a process (re)joining a given group. A library for atomic state transfer as provided in Isis [3] would have again simplified building these fault-tolerant programs. Wood discusses building fault-tolerant applications for Amoeba in more detail [35].

Our decision to make the group primitives blocking and to achieve parallelism through running multiple threads per process has forced us to write cleanly-structured applications. Per thread it is easy to reason how the application will behave and activities that can be performed concurrently can be easily expressed by starting a thread for them. We believe that many applications would have been more difficult to write if the group primitives had been nonblocking and parallelism was achieved by overlapping communication and synchronization using a single thread. In some cases the overhead of starting a thread was too high and the performance could have benefited from nonblocking primitives, but we believe that the problem is better solved by optimizing the performance of the thread package than by reducing the ease of programming. Similar observations have been made for RPC systems [6].

Unfortunately the decision to have multiple threads per process and nonblocking group primitives sometimes made it hard to port the group system to other existing systems based on a different model. For example, many UNIX systems do not have kernel threads and therefore a blocking operation results in the whole process being blocked. These problems can be circumvented on most UNIX systems by using the *ioctl* and *select* system calls, but these are hard to program and sometimes do not perform well.

We decided to implement the group protocols in the kernel, because we believed that the implementation should perform well in order to attract applications. It is unclear whether this was the right decision. Recently Oey et al. reported on running the protocols in user space [23]. They measured a 32% performance decrease in communication performance for synthetic benchmarks, but for most applications the performance decrease was very small. In addition, recent work by Thekkath et al. [32], and Maeda and Bershad [20] shows how good performance can be obtained by carefully dividing the communication functionality between a user server and application library. It should be noted that at the time we implemented the Amoeba protocols these results were unknown.

By moving the code out of the kernel into servers and application libraries, we could have separated out the communication functionality more cleanly in modules. For example, the failure detection in the current system is intertwined with the protocol code for sending and receiving messages. In addition, the RPC module performs its own failure detection. We should have put this functionality in a separate module so that we could have reasoned about it independently of the rest of the system. The failure detection and group rebuilding code turned out to be the hardest parts of the system to get correct. Newer versions of Isis [26] and more recent systems such systems such Transis [1] separate these pieces of functionality cleanly.

6 Related Work

In this section we will compare Amoeba with other complete group communication packages and their protocols; a detailed comparison of our reliable broadcast protocol with other protocols can be found in [14].

The first system supporting group communication, described in [8], is the V system. It 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 [22] runs on top of V).

The protocols in our systems were influenced by Chang and Maxemchuck (CM), who describe a family of broadcast protocols [7]. 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 larger than 2 in the normal case. 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, unlike ours, physical broadcast instead of multicast for all messages and is restricted to a single LAN.

The protocols that are used in the first complete system supporting ordered group communication, described in [4], 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 in the normal (no failure) case. 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. The primary difference is that Isis emphasizes fault tolerance where as our work emphasizes high performance.

Recently the protocols for Isis have been redesigned [5]. 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 on whether 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. 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 [33].

Amir et al. describe a recently-built system, called Transis, that supports a number of protocols with varying properties [1]. 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 active 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.

In [24] 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 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.

7 Conclusion

We reported on the performance and the experience with the Amoeba group communication system and its protocols. An in-kernel implementation of these protocols achieves high performance. 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.

Based on our experience with implementing these protocols and their usage in various applications we have learned that: (1) the scalability of our sequencer-based protocols is limited by message processing time. Promising techniques for overcoming this problem are (optimistic) active messages and dynamic sequencer-based protocols; (2) the flexibility and modularity of user-level implementations of protocols is likely to outweigh the potential performance loss.

Acknowledgments

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, Ken Birman, Mootaz Elnozahy, Barbara Liskov, Robbert van Renesse, Kees Verstoep, Mark Wood, and Willy Zwaenepoel for providing comments on earlier versions of this paper.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proc. 22nd Int'l Symp. on Fault-Tolerant Computing*, pages 76–84, Boston, MA, June 1992.

[2] R. Bhoedjang, T. Ruhl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Proc. of Third Symp. on Experiences with Distributed and Multiprocessor S*, pages 213–226, San Diego, CA, Sept 1993.

[3] K.P. Birman, R. Cooper, T.A. Joseph, K.P. Kane, F. Schmuck, and M. Wood. Isis - a distributed programming environment. Technical Report User's Guide and Reference Manual, Cornell University, Ithaca, NY, June 1990.

[4] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comp. Syst.*, 5(1):47–76, Feb. 1987.

[5] K.P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, Aug. 1991.

[6] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Trans. Comp. Syst.*, 2(1):39–59, Feb. 1984.

[7] J. Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comp. Syst.*, 2(3):251–273, August 1984.

[8] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernel. *ACM Trans. Comp. Syst.*, 3(2):77–107, May 1985.

[9] S.E. Deering and D.R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comp. Syst.*, 8(2):85–110, May 1990.

[10] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[11] V. Hadzilacos and S. Toueg. *Distributed Systems 2nd ed.*, chapter Fault-Tolerant Broadcasts and Related Problems. Addison-Wesley, Reading, MA, 1993.

[12] H-P. Heinzle, H.E. Bal, and K.G. Langendoen. Implementing object-based distributed shared memory on transputers. In *World Transputer Congress 1994*, pages 390–405, Lake Como, Italy, Sept. 1994.

[13] W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, D.A. Wallach, and W.E. Weihl. Efficient implementation of high-level languages on user-level communication. Technical Report MIT/LCS/TR-616, Cambridge, MA, May 1994.

[14] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1992.

[15] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Proc. Symp. on Experiences with Distributed and Multiprocessor Systems II*, pages 297–312, Newport Beach, CA, March 1992.

[16] M.F. Kaashoek and A.S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proc. Eleventh Int'l Conf. on Distributed Computing Systems*, pages 222–230, Arlington, TX, May 1991.

[17] M.F. Kaashoek, A.S. Tanenbaum, S. Flynn Hummel, and H.E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–20, Oct. 1989.

[18] M.F. Kaashoek, A.S. Tanenbaum, and K. Verstoep. Using group communication to implement a fault-tolerant directory service. In *Proc. 13th Int'l Conf. on Distributed Computing Systems*, pages 130–139, Pittsburgh, PA, May 1993.

[19] M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum. Flip: an internetwork protocol for supporting distributed systems. *ACM Trans. Comp. Syst.*, 11(1):73–106, Feb. 1993.

[20] C. Maeda and B. Bershad. Protocol service decomposition for high performance networking. *14th ACM Symp. on Operating Systems Principles*, pages 244–255, Dec. 1993.

[21] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[22] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *Proc. Eighth Int'l Conf. on Distributed Computing Systems*, pages 439–446, San Jose, CA, June 1988.

[23] M. Oey, K. Langendoen, and H.E. Bal. Comparing kernel-space and user-space communication protocols on Amoeba. In *Proc. of 15th Int'l Conf. on Distributed Computing Systems*, pages 238–246, Vancouver, Canada, May 1995.

[24] L.L. Peterson, N.C. Buchholtz, and R.D. Schlichting. Preserving and using context information in IPC. *ACM Trans. Comp. Syst.*, 7(3):217–246, Aug. 1989.

[25] J. Postel. Internet protocol. Technical Report RFC 791, SRI Network Information Center, Sept. 1981.

[26] A.M. Ricciardi and K.P. Birman. Using process groups to implement failure detection in asynchronous environme. In *Proc. of the Tenth ACM Symp. on Principles of Distributed Computing*, pages 341–351, Quebec, Canada, 1991.

[27] F.B. Schneider. Byzantine generals in action: Implementing fail-stop processes. *ACM Trans. Comp. Syst.*, 2(2):145–154, May 1984.

[28] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[29] A.S. Tanenbaum. *Computer Networks 2nd ed.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[30] A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, Aug. 1992.

[31] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S.J. Mullender, A. Jansen, and G. van Rossum. Experiences with the amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, Dec. 1990.

[32] C.A. Thekkath, T.D. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proc. of Int'l Conf. on Communications Architectures, Protocols and Applicatio*, pages 64–73, San Francisco, CA, 13-17, 1993.

[33] R. van Renesse, K.P. Birman, R. Cooper, B. Glade, and P. Stephenson. A RISC approach to process groups. In *Proc. Usenix Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, April 1992.

[34] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proc. of 5th Symp. on Principles and Practice of Parallel Programming*, pages 217–226, Santa Barbara, California, July 1995.

[35] M.D. Wood. Replicated RPC using amoeba closed group communication. In *Proc. of the 13th Int'l Conf. on Distributed Computing Systems*, pages 499–507, Pittsburgh, PA, May 1993.

[36] M. Young, A. Tevenian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. Duality of memory and communication in the implementation of a multiprocessor. In *Proc. Eleventh Symp. on Operating Systems Principles*, pages 63–67, Austin, TX, Nov. 1987.