A performance comparison of asynchronous atomic broadcast protocols

Distrib. Syst. Engng 1(1994)177–201

A performance comparison of asynchronous atomic broadcast protocols

Flaviu Cristian, Richard de Beijer and Shivakant Mishra

Department of Computer Science & Engineering, University of California, San Diego, La Jolla, CA 92093-0114, USA

Received 19 August 1993

Abstract. Atomic broadcast ensures that concurrent updates to replicated data maintained by a process group are consistently delivered to all group members despite random communication delays and failures. By simplifying the programming of applications that use replicated data, atomic broadcast provides basic support for implementing fault-tolerance in distributed systems. This paper reports discrete event simulation results that compare the performance of four asynchronous atomic broadcast protocols. We investigate five performance indexes: average delivery time, average stability time, average number of physical messages sent per update broadcast, maximum buffer size, and distribution of processing load among group members. These indexes are measured as a function of group size and update interarrival time, both in the absence of failures and in the presence of a single communication failure. Our comparison shows that there is no overall best protocol. We identify those application areas where a protocol dominates the other protocols and we discuss some protocol design techniques for achieving good performance.

1. Introduction

Process groups are an important structuring tool for implementing fault-tolerant services. The underlying idea is that members of a process group replicate service state, so that if some members fail, the surviving members know enough to continue to provide the specific service. *Atomic broadcast* is a general group communication service that can be used by group members to disseminate replicated state updates. The properties of an atomic broadcast service depend on whether communication delays are bounded or unbounded. When they are bounded, the service is called synchronous [4]. In this paper we are interested in *asynchronous* atomic broadcast which does not assume any bounds on communication delays. For brevity, we refer to an asynchronous atomic broadcast service simply as a broadcast service in what follows. For a fixed application process group, a broadcast service ensures that all correct members receive all broadcast updates in the same order: if p and q are arbitrary members of the same group, hist(p) and hist(q) are the histories of updates delivered to p and q by the broadcast service since group creation, then either hist(p) is a prefix of hist(q) or hist(p) = hist(q) or hist(q) is a prefix of hist(p) [3].

The *broadcast delivery time* is defined to be the duration between the moment an update is entrusted to the broadcast service by a group member and the moment the update is delivered by the service to every group member. When messages can get lost, the broadcast servers that implement the distributed broadcast service must store a message that is being broadcast in a local buffer until they learn that the message is *stable*, that is, it was delivered to all group members. The *broadcast stability time* is the duration between the moment a broadcast server receives an update to be broadcast and the moment all broadcast servers learn that the update is stable.

The goal of our study was to compare the performance of four atomic broadcast protocols [1–3, 5] that belong to two broad classes: sequencer-based and train protocols. In a sequencer-based protocol, a unique broadcast server, the *sequencer*, imposes a total ordering on all updates originating from all broadcast servers. Positive [2] or negative [1, 5] acknowledgments are used to ensure that broadcast updates are received by all group members. In a train protocol [3], there is a cyclic order among broadcast servers and a train of updates circulates between them in this order. If a server wants to broadcast updates, it waits for the train, appends the updates at the end of the train, lets the train move around once, and then purges the updates from the train. The order in which group members receive broadcast updates is the order in which they board the train.

While some of the above protocols have been implemented and their performance has been reported, it is difficult to evaluate them comparatively based on such information, as each of them has been implemented in a

different computing environment and under different assumptions. Our goal is to simulate these protocols in a common computing environment and compare their relative performance under the same assumptions. In addition to a comparison that is fair, such an exercise also yields insight into what features are useful in designing broadcast protocols that perform well under different conditions. Since communication delays are unbounded, the delivery and stability times are unbounded. Thus, our interest will be in comparing average delivery and stability times, the average number of messages needed per broadcast, the distribution of processing load among group members (in terms of the number of messages processed) and the maximum buffer size (in terms of the number of messages) needed by each of these protocols. We are interested in both failure free performance, as well as performance in the presence of communication failures. This allows us to determine the jitter introduced by an occasional failure in the behaviour of the protocols.

We start by stating the assumptions used in our protocol behaviour simulations. We then briefly describe the four protocols we have simulated. The results of our simulation are explained in sections 4, 5, 6 and 7. Section 8 describes some useful techniques to improve performance of a broadcast protocol as observed from our simulation, and section 9 concludes the paper.

2. Assumptions

We consider a single group of broadcast servers running on distinct processors in a point-to-point network. The servers disseminate updates on account of higher level applications running on these processors. Communication is via a datagram service that has omission/performance failure semantics: messages can get lost or be late, but corruption of their contents is unlikely. Since we are not interested in the group structure of the applications that use the broadcast service, we will refer to the group of broadcast servers implementing the broadcast service as 'the group'.

We make use of *discrete event simulation* [6] to explore the behaviour of broadcast protocols. This yields a fair comparison between them, since their performance is measured under identical conditions. For failure free broadcasts, we simulated the sending of ten thousand updates, so that all group members process the same number of updates during the simulation. For the simulation of broadcasts in the presence of a single communication failure, the negative acknowledgment protocols require the simulation of a few more successful broadcasts than the positive acknowledgment based protocols, in order to let group members detect the communication failure affecting the 'last' broadcast.

The following assumptions are made in our simulation study.

Since the focus is on the evaluation of broadcast, not membership protocols, we assume a constant group membership throughout any simulation run. In other terms, we assume that no server crash and join events occur during simulation runs.

Communication delays between group members are assumed to follow the distribution shown in figure 1. This distribution has been obtained empirically by measuring round trip delays for a short message sent over an Ethernet between two Unix processes over a period of 48 hours. The two SunOs processes were running on a Sun4 and IPX SparcStation, respectively. The communication delay measured is the *end-to-end* delay: it includes process switching time, time spent in the sending and receiving UDP/IP layers as well as the (much smaller fraction of) time spent in the Ethernet adapter cards and the Ethernet itself. The mean value of the process-to-process communication delays was 3.318 milliseconds and the minimum observed was 2.181 milliseconds.. Although the observed maximum delay (over 25 seconds) was way to the right of the tail of the density graph in figure 1, the 99% timeout was 14.5 milliseconds (that is, only 1% of messages experienced delays greater than 14.5 milliseconds).

The simulations assume that the interarrival time between update arrivals at broadcast servers is *exponential* with mean expected value $1/\lambda$. We have considered values of $1/\lambda$ equal to 15.0, 25.0, 50.0, 75.0, 100.0, 200.0, 300.0, and 400.0 milliseconds. We have considered group sizes n between 2 and 10.

The simulations further assume that the CPU time needed by a broadcast server to process the sending or receipt of an update is negligible, i.e. null, compared to the broadcast server to broadcast server communication delay. For the group sizes we have considered, and for current CPU speeds of 10 Mips and up, this time is of the order of tens of microseconds, negligible compared to the server to server communication delays of the order of thousands of microseconds (i.e. milliseconds). Our assumption that the contribution of server CPU time to total update delivery time is negligible remains valid as a first order approximation of the delivery delay that would

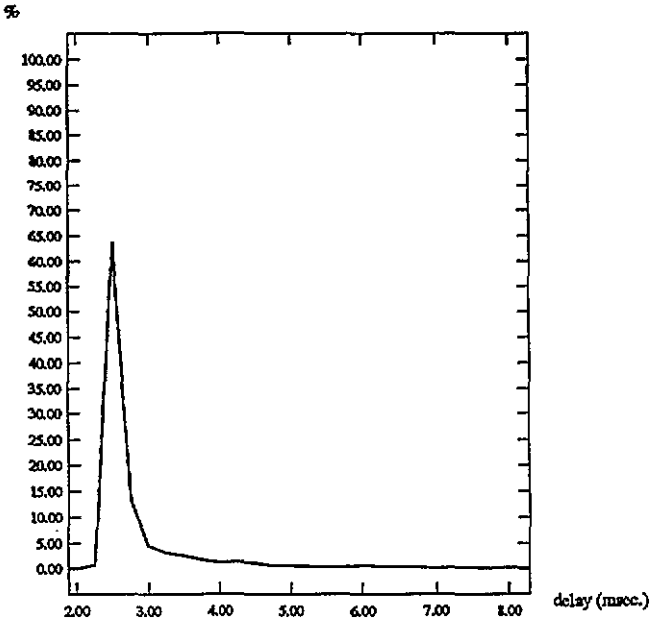

Figure 1. Communication delay density.

be observed in a real system for as long as the CPU update processing time represents less than 10 per cent of the per member update interarrival time, that is $(1/\lambda)/n > 1$ msec. This assumption is satisfied for all update interarrival times and group sizes considered by our simulations.

We simulate the sending of an update to $n-1$ group members as the sending of $n-1$ point-to-point messages. Alternatively, this could be done by some other method such as sending point-to-point messages along a spanning tree. We chose the former method because our goal is to measure overall performance in a generic communication environment. The CPU processing time for sending $n-1$ messages can become substantial as n increases, for example for groups with hundreds of members. To avoid invalidating our null CPU time assumptions, we restricted our study to small group sizes. We have simulated the behaviour of the atomic broadcast protocols both in the absence of failures and in the presence of one communication failure per broadcast. The communication failure is simulated as follows. When an update broadcast is initiated, one of the messages to be sent on account of the broadcast is chosen to be lost. The choice is made in such a way that the probability of the loss of any given message among those to be sent is the same. To simulate the message loss, the randomly chosen message is simply not sent.

3. Overview of broadcast protocols

This section gives a basic overview of the protocols in chronological order of publication.

We first started out with the simulation of the Tandem global update protocol [2]. However, the preliminary results showed that the performance of this protocol is notably worse than that of the other protocols, since it allows at most one update to be broadcast at a time per group. So, we investigated a variant, to be called the *Positive Acknowledgment* or PA protocol, that allows concurrent broadcasts. In the PA protocol, a sender s initiates the broadcast of an update u by sending a message containing u and a local sequence number l to the sequencer. If the previous message received from s by the sequencer had local sequence number $l-1$, the sequencer attaches a global sequence number n to this update and sends messages (u,n) to every group member. If the (u,l) message is out of order, the sequencer stores (u,l) in a local buffer until the previous message $(u',l-1)$ arrives from s and then globally orders and broadcasts u' and u consistently with their origination order at s. Upon receipt of (u,n), each member sends a positive acknowledgment for n to the sequences. Message losses are detected by timeouts, and result in message retransmissions; the timeout is set to four times the average message communication delay in our experiments (this corresponds to the 98.9% timeout delay). Updates are delivered at members in the order imposed by the sequence numbers attached by the sequencer. Concurrency is allowed among broadcast servers as well as among multiple broadcast requests at a single server. Update stability is determined as follow. Group members piggyback on their messages to the sequencer the sequence number *ld* of the last update they have delivered. The sequencer records them in an array with one entry per member and piggybacks the minimum sequencer number *ld_all* stored in the array on all messages it sends to group members. Thus, an update u with sequence number n is stable when all members have received from the sequencer messages with piggybacked stability number *ld_all* at least n.

The broadcast protocol of Amoeba [5] is also sequencer-based. To initiate the broadcast of update u a sender sends a message containing u to the sequencer. The sequencer attaches a sequence number n to u and sends messages (u,n) to all group members. However, unlike in the positive acknowledgment protocol, group members do not send any (positive) acknowledgment back when they receive such messages. After having received a message (u,n), a member sends a negative acknowledgment to the sequencer only if the next message it receives contains a sequence number greater than $n+1$. The sequencer re-transmits messages only upon receiving such negative acknowledgments. Concurrency is allowed among distinct broadcast servers, but in the protocol described in [5] each broadcast server handles only one broadcast request at a time. The stability of a broadcast is established in the same manner as for the positive acknowledgment protocol.

In the train protocol [3], there is a cyclic order among group members. A train containing a sequence of updates circulates from one member to another in this order. A member (the sender) that wants to broadcast an update waits for the train to arrive. When the train arrives, the sender first delivers all updates carried by the train, and then appends all updates that it wants to broadcast at the end of the train. The sender removes these updates when he sees the train again. If there are no broadcasts in progress, the empty train remains idle at some designated group member, *the trainmaster*, and in such a case a sender must request the train in order to broadcast an update. A lost train is detected by the trainmaster based on a timeout mechanism; this timeout is set to $4 * N *$ average communication delay in our experiments. The trainmaster is also responsible for regenerating a lost train. Stability of an update is established when the train completes one more round after delivering the update to all group members.

The atomic broadcast protocol (ABCAST) of Isis [1] is sequencer-based. To broadcast an update u, a sender s causally broadcasts messages containing the update u and a local vector of sequence numbers—or timestamp vector—l to all group members. When the sequencer receives (u,l,s), it assigns to this update a new global sequence number n and causally broadcasts (n,l,s) to all members. After receiving a message carrying an update (u,l,s), a member waits to get from the sequencer the final sequence number n for the (l,s) update. Group members use the global ordering imposed by the sequencer to deliver updates. The causal broadcast protocol used by the atomic broadcast protocol uses a lower level first-in-

first-out (FIFO) transport protocol to send messages between pairs of broadcast servers, so that messages received by a destination process are received in the order sent by the sender process. This FIFO protocol makes use of low level sequence numbers and negative acknowledgments to detect lost messages. Stability of an update u identified by (l,s) is established at a group member when it has received from every other group member s', a message (u',l',s'), such that the vector timestamp l' reflects the delivery of update u and s' (i.e., $l' > l$). Since this paper investigates only atomic broadcast protocols, when we talk about the Isis protocol, we mean the ABCAST Isis protocol.

Notice that in the PA, Amoeba, and Isis protocols, it is possible for the sequencer to batch the ordering information for several messages: the sequencer could wait until k updates arrive, where $k > 0$, before sending the ordering information for all these updates in a single message. The descriptions of both the Amoeba and Isis protocols mention this possible optimization. For reasons of simplicity, and because the descriptions of these protocols do not clearly state what the parameter k should be or how long the sequencer should wait before sending the ordering information, we chose to simulate the sending of a separate ordering message for each update to be ordered. We expect that if the above optimization is applied, it will result in larger delivery, stability and message loss detection times, and fewer number of message exchanges per update broadcast than that reported in this paper.

4. Average delivery time

4.1. Failure-free performance

4.1.1. The PA protocol. The average broadcast delivery time of the PA protocol is shown in figure 2 as a function of group size, and in figure 3 as a function of mean update interarrival time. There are three delay components that contribute to the delivery time: (1) the message

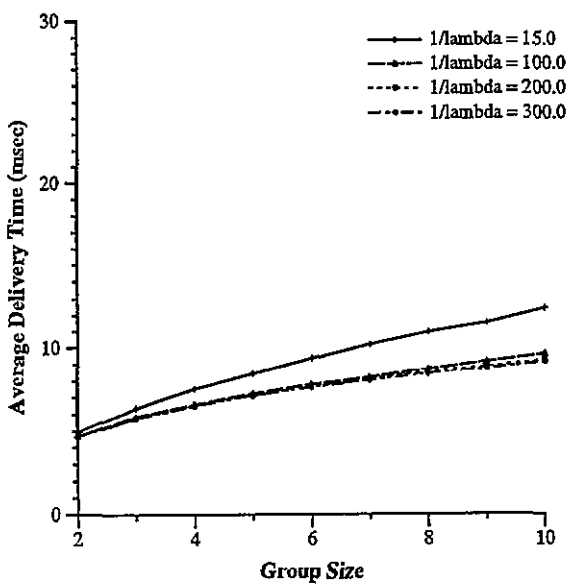

Figure 2. Delivery time (PA).

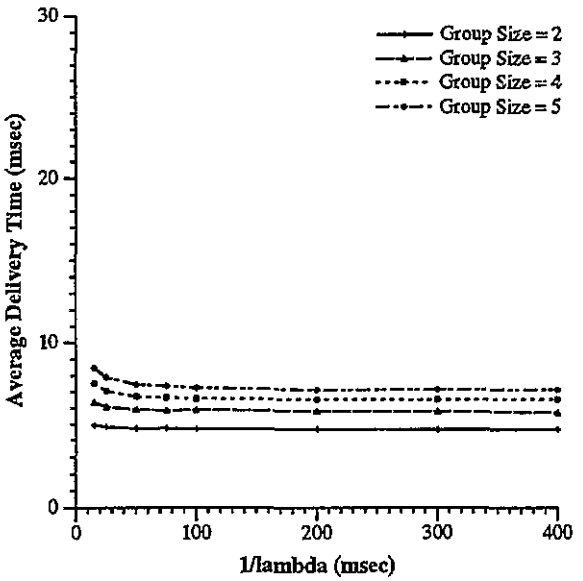

Figure 3. Delivery time (PA).

communication delay from the sender to the sequencer, (2) the time an update may wait at the sequencer, because the sequencer hasn't yet received some earlier update from the same sender, and (3) the message communication delay from the sequencer to the group members. Only the third delay component affects the delivery time if the sender is the sequencer itself. There are two reasons for the increase in the average delivery time with the group size. First, since in our simulation every group member broadcasts an equal number of updates, the percentage of the number of updates broadcast by the sequencer decreases as the group size increases. Since the delivery time for an update broadcast by the sequencer is less than the broadcast by another group member, the overall average delivery time increases with the group size. Second, the third delay component depends on the largest communication delay experienced among all group members. This increases when the group size increases, and as a result the average delivery time increases. The increase in the average delivery time is higher for lower mean update interarrival times, because the chances of messages arriving out of order at the sequencer are high when the updates are generated rapidly. This increases the second delay component. This is also the reason for the higher delivery times observed when the mean interarrival times decrease in figure 3. The minimum broadcast delivery time was observed to be 2.25 milliseconds, which corresponds to a broadcast done by the sequencer.

4.1.2. The Amoeba protocol. The average delivery time of an update in the Amoeba protocol is shown in figure 4 as a function of group size, and in figure 5 as a function of mean interarrival time. The delay components that contribute to the delivery time are (1) the time an update must wait at the sender process if the sender is waiting for completion of an earlier broadcast, (2) the message communication delay from the sender to the sequencer, and (3) the message communication delay from the sequencer to the group members. The dependency between the average delivery time and group size and mean interarrival times is

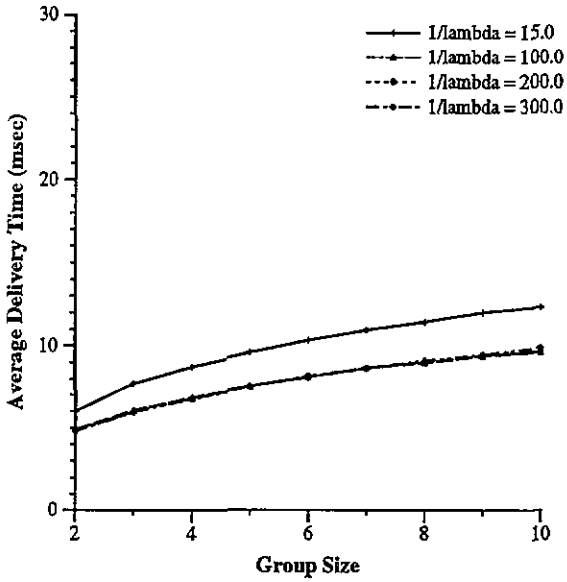

Figure 4. Delivery time (Amoeba).

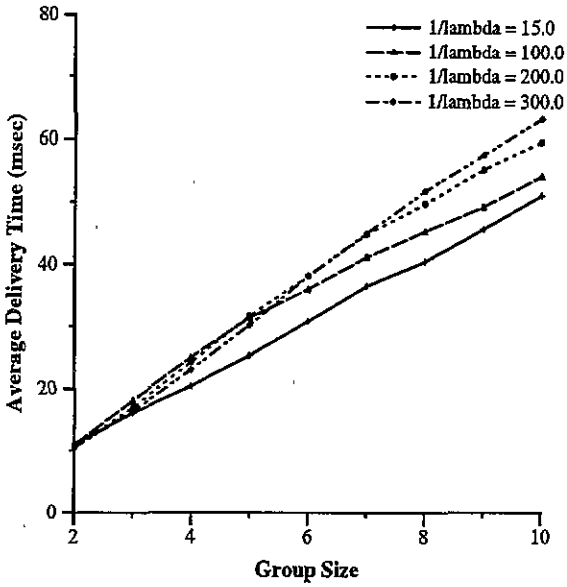

Figure 6. Delivery time (train).

similar to that seen for the PA protocol. The delivery time is higher than for the PA protocol for lower mean interarrival times because the likelihood that an update has to wait at the sender is higher when the update interarrival rate increases.

4.1.3. The train protocol. The average broadcast delivery time of the train protocol is shown in figure 6 as a function of group size, and in figure 7 as a function of mean update interarrival time. There are two significant delay subcomponents: (1) the time a sender has to wait between an update arrival and the train arrival and (2) the time taken by the train to complete one round after the update was appended to it.

The average broadcast delivery time increases almost linearly with the group size because these two delay components increase with the group size. This dependency, however, is not completely linear, since the train may be idle when the update is generated and this may cause the sender to wait for a longer time (until a timeout) before requesting, and finally receiving the train. The average delivery time first increases with the mean update interarrival time reaching a maximum value, after which it decreases, albeit slightly. The initial increase is due to a decrease in the number of updates carried by the train when the mean update interarrival time increases, which in turn increases the average delivery time. The delivery time reaches a maximum when the train carries at most one update at any time. A slight further decrease for larger mean update interarrival times is caused by the following fact. In order to broadcast an update, a sender first waits for the train for a fixed amount of time since it last saw it. Only if the sender does not see the train arriving during this time does it send a train request. So the first delay component consists of the difference between the moment the update arrives and the time a request for the train is initiated. This difference decreases as the mean update interarrival time increases.

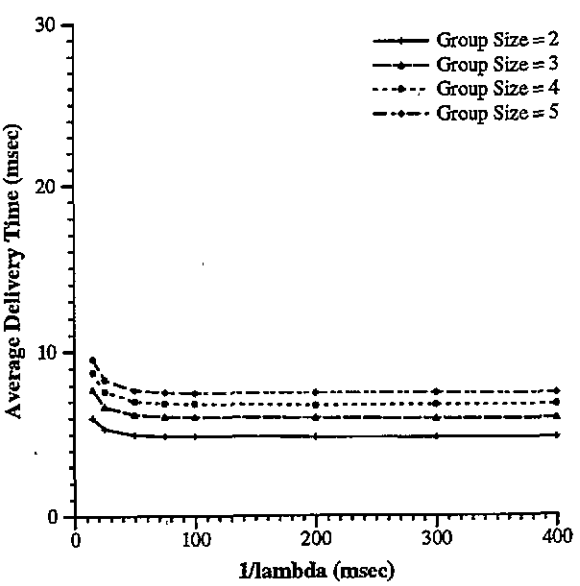

Figure 5. Delivery time (Amoeba).

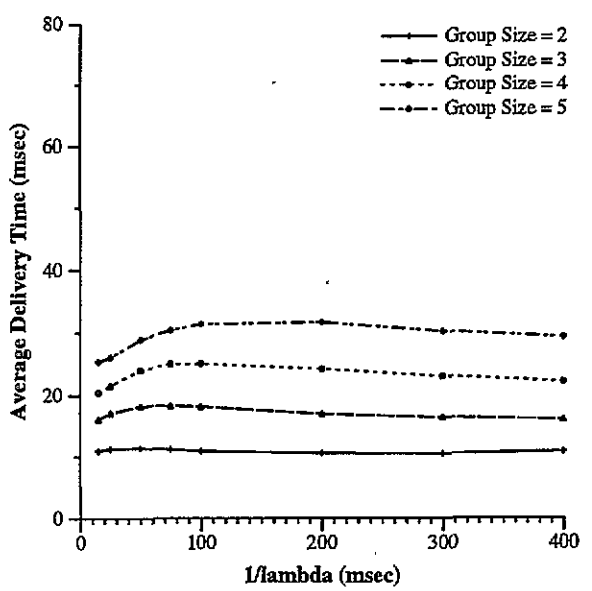

Figure 7. Delivery time (train).

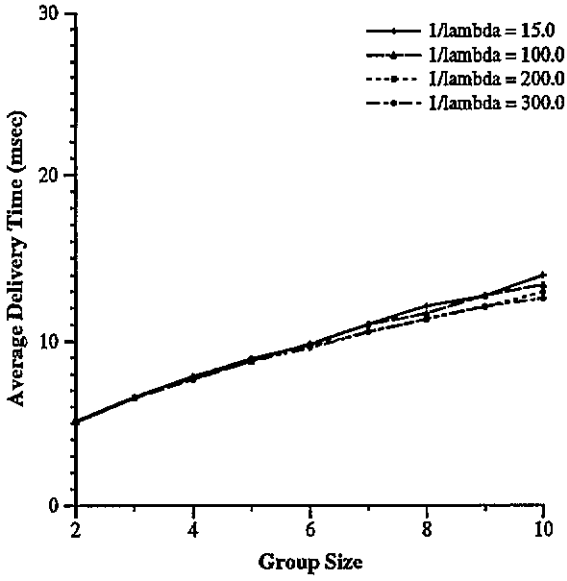

Figure 8. Delivery time (Isis).

4.1.4. The Isis protocol. The average broadcast delivery time of an update is shown in figure 8 as a function of group size, and in figure 9 as a function of mean update interarrival time. There are two delay components in the Isis broadcast delivery time: (1) the communication delay to send a causal broadcast from the sender to all group members, and (2) the communication delay to causally send the ordering information from the sequencer to all groups members. The dependency between the average delivery time and group size or mean interarrival time is similar to that seen in the PA and the Amoeba protocols. The reason for this is that all these protocols are sequencer based with similar delivery mechanism in the absence of failures.

4.2. Delivery time in the presence of a communication failure

4.2.1. The PA protocol. The average delivery time in the presence of one communication failure per broadcast is

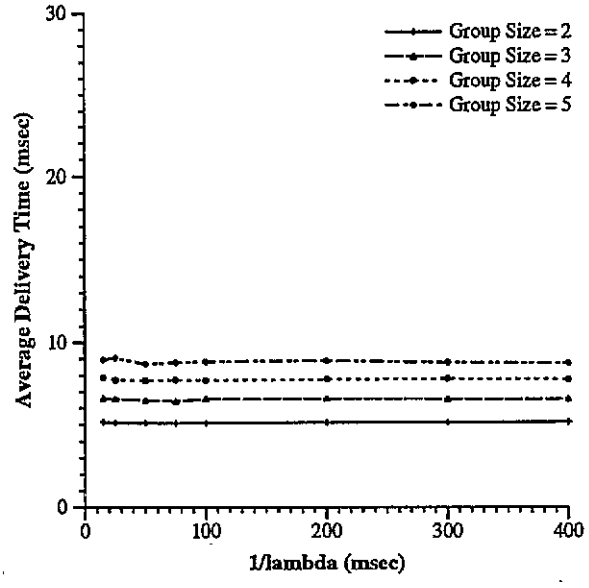

Figure 9. Delivery time (Isis).

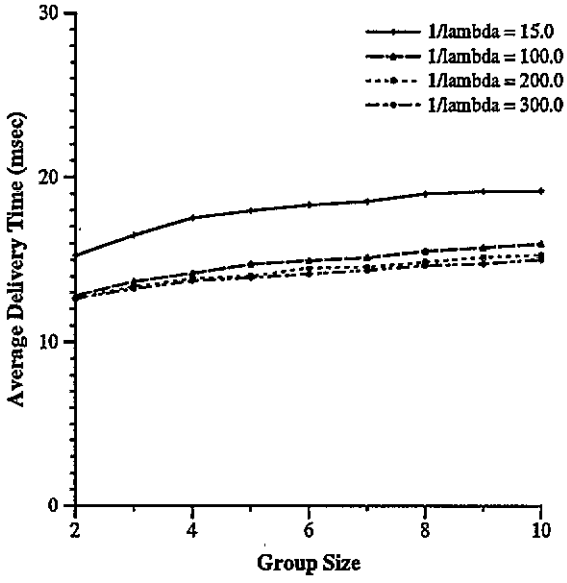

Figure 10. Delivery time; 1 msg loss per broadcast (PA).

shown in figure 10 as a function of group size and in figure 11 as a function of mean update interarrival time. A message loss in this protocol is detected by timeout. The timeout interval is independent of the mean update interarrival time or the group size. As a result, the dependency between average delivery time and group size, and average delivery time and mean update interarrival time are similar to those in the failure free case: the average delivery time is essentially increased by the timeout delay compared to the failure free case.

4.2.2. The Amoeba protocol. Figure 12 shows the average broadcast delivery time as a function of group size in the presence of one communication failure. The average broadcast delivery time becomes very large for lower group sizes and it reduces when the group size increases. The reason for this is that a message loss in this protocol is detected by some member when it receives a message that follows the lost message. When the group size is small, the per member update interarrival time at the sequencer is

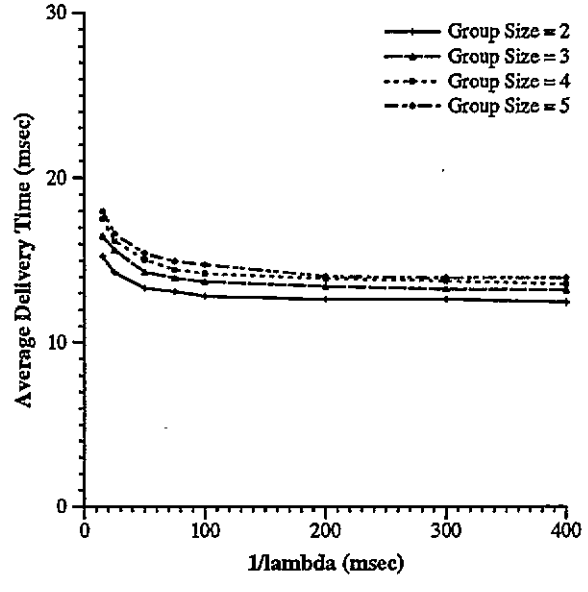

Figure 11. Delivery time; 1 msg loss per broadcast (PA).

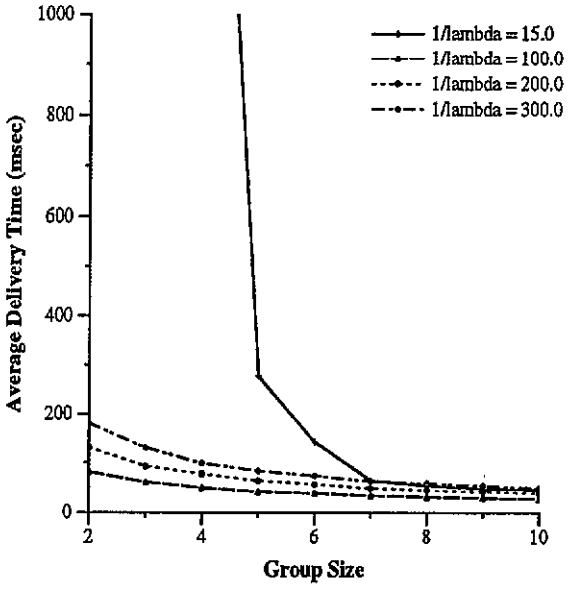

Figure 12. Delivery time; 1 msg loss per broadcast (Amoeba).

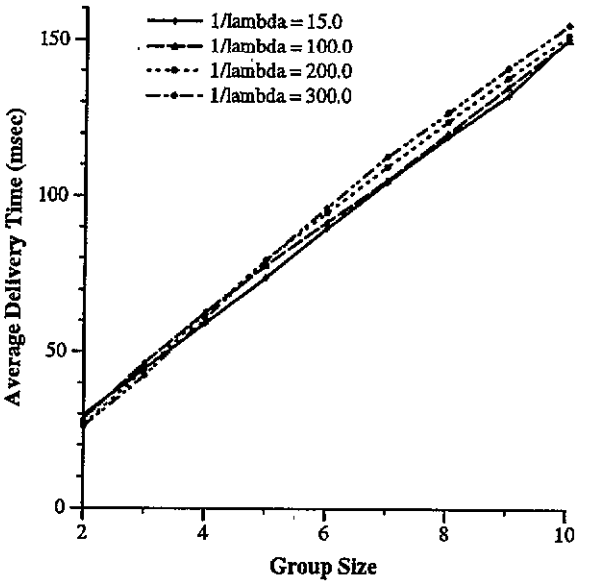

Figure 14. Delivery time; 1 msg loss per broadcast (train).

large, so the 'following' message arrives after a longer time. As a result it takes longer to detect a message loss. Furthermore, for smaller mean interarrival times (15.0 ms) and smaller group sizes, the first delay component, i.e. the delay at the sender, is significantly higher because of the longer time needed to detect message losses. This causes the delivery time to be extremely high for low mean interarrival times and low group sizes. Figure 13 shows the average broadcast delivery time as a function of the mean interarrival time in the presence of one communication failure. The average broadcast delivery time is very large for lower interarrival times because of the first delay component. When the mean update interarrival time is small, the likelihood that an update has to wait at the sender is high (particularly because a message loss increases the delivery time). The delivery time reduces to a minimum when the mean interarrival time is about 50 ms, at which time the first delay component reduces to a minimum. After that, the average delivery time increases with the mean interarrival time. This increase is due to an increase in the time needed to detect a message loss when the mean interarrival times increases.

4.2.3. The train protocol. The dependencies between average delivery time and group size, and average delivery time and mean update interarrival time in the presence of one communication failure per broadcast are similar to the failure free case, as shown by the figures 14 and 15. Like in the PA protocol, the average broadcast delivery time is increased by the timeout delay (for detecting the loss of the train).

4.2.4. The Isis protocol. The average delivery time in the presence of one communication failure per broadcast is shown in figure 16 as a function of group size and in figure 17 as a function of mean update interarrival time. A message loss is detected by a group member when it receives another message with a higher transport level

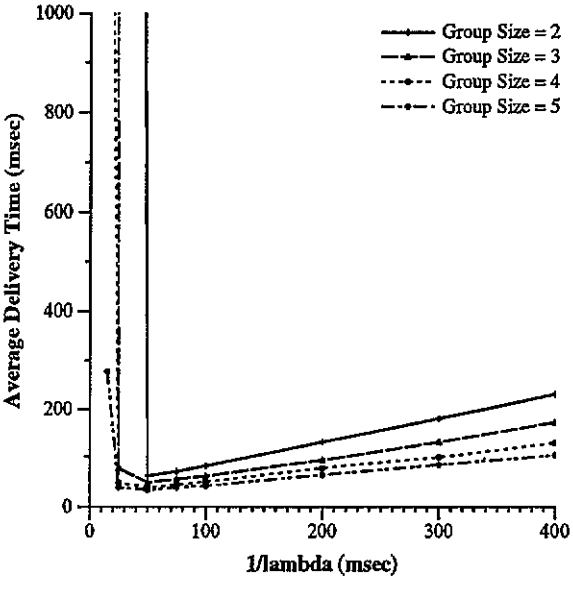

Figure 13. Delivery time; 1 msg loss per broadcast (Amoeba).

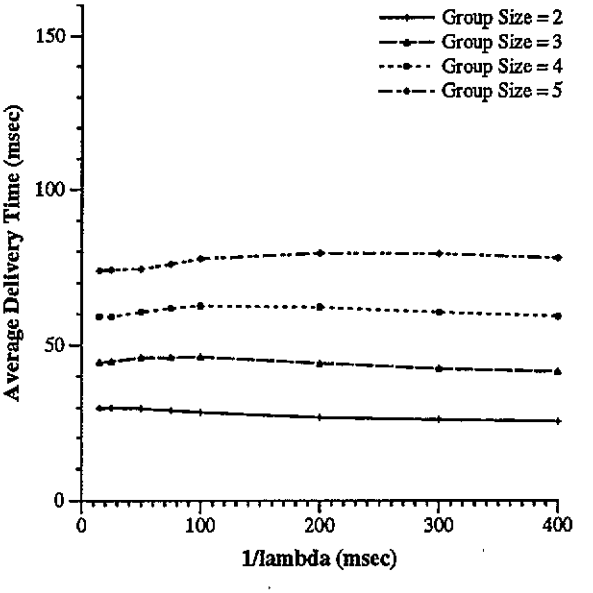

Figure 15. Delivery time; 1 msg loss per broadcast (train).

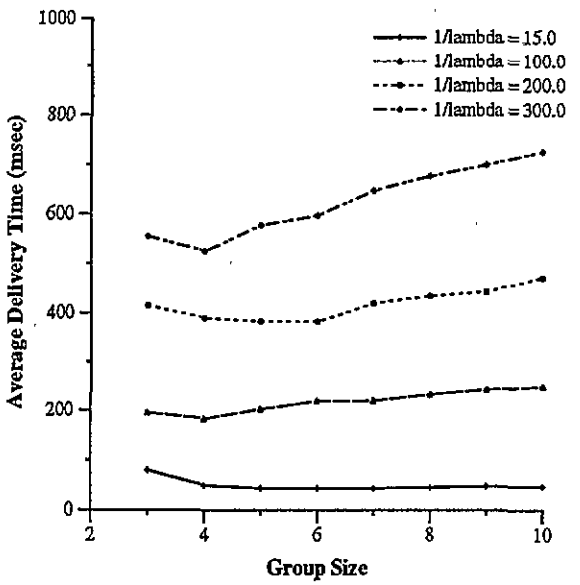

Figure 16. Delivery time; 1 msg loss per broadcast (Isis).

sequence number. This causes the average delivery time to grow as the mean update interarrival time grows (figure 17). The average delivery time increases when the group size increases beyond 3 because the percentage of the number of updates broadcast by the sequencer decreases, and the second Isis delivery delay component increases. The average delivery time is slightly larger, when the group size is small. Indeed, when the group size is small, the likelihood of losing two or more consecutive messages from the same sender is larger, and hence, the message loss detection time is larger. We did not plot the average delivery times for group size 2, because they were very large.

4.3. Comparison

The failure-free average delivery times of all protocols are compared in figures 18 and 19 as a function of group size and in figures 20 and 21 as a function of mean update interarrival time. The average delivery times of the PA,

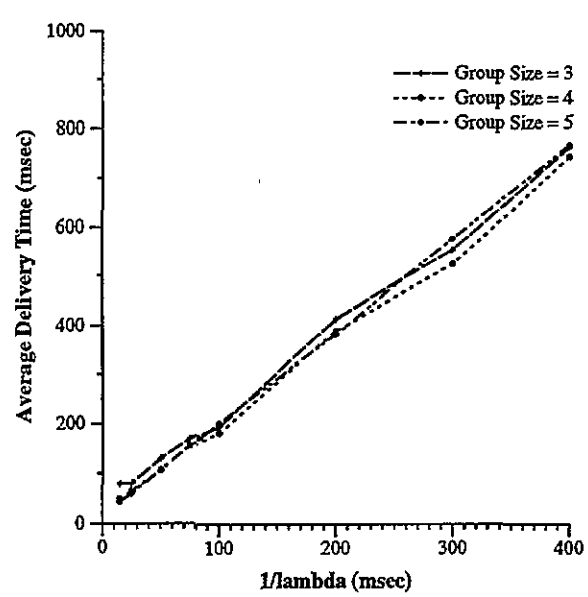

Figure 17. Delivery time; 1 msg loss per broadcast (Isis).

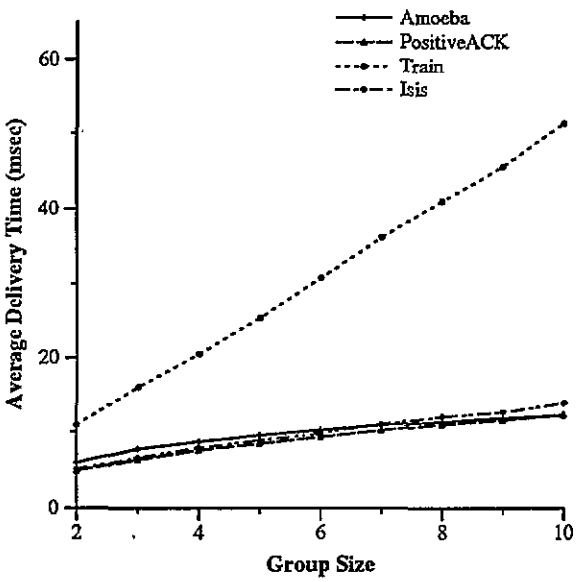

Figure 18. Comparison for $1/\lambda = 15.0$ ms; no message loss.

Amoeba and Isis protocols are similar, while those of the train protocol are higher and increase linearly with the group size. The delivery times of the PA, Amoeba and Isis protocols are lower because these protocols allow for more concurrency by enabling multiple group members to send messages in parallel. For example, the sequencer can send messages to group members while a group member sends a message to the sequencer. The train protocol sends at most one message at any time, and this contributes to an increase in the protocol delivery time. However, as we will see later, this characteristic of the train protocol is also one of its strengths: low number of messages per broadcast at high update arrival rates.

Although the PA, Amoeba and Isis protocols are all sequencer based, the average delivery times of Isis are slightly higher than those of the PA and Amoeba protocols. The reason for this is that Isis uses a lower level causal order broadcast protocol to send messages. This causal order broadcast introduces some additional restrictions on concurrency that are not present in the other two

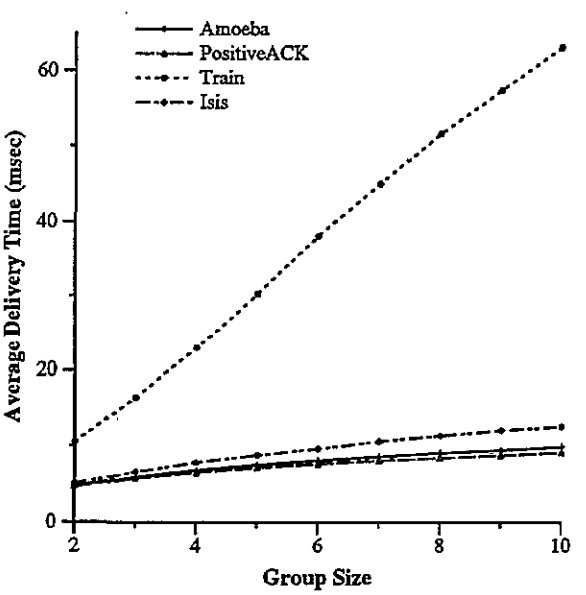

Figure 19. Comparison for $1/\lambda = 300.0$ ms; no message loss.

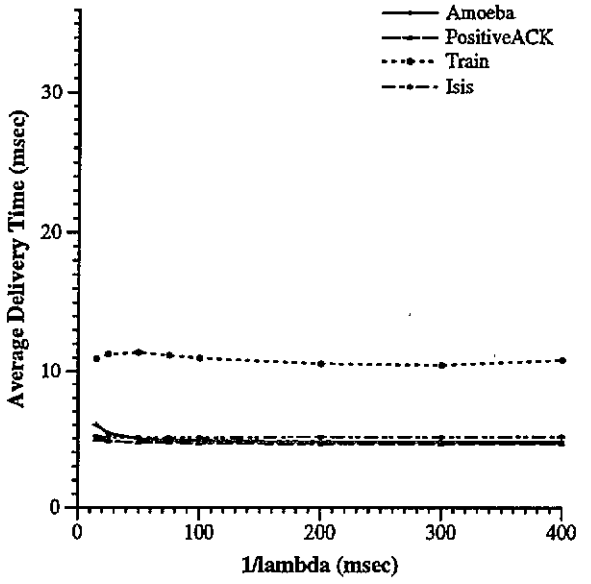

Figure 20. Comparison for group size 2; no message loss.

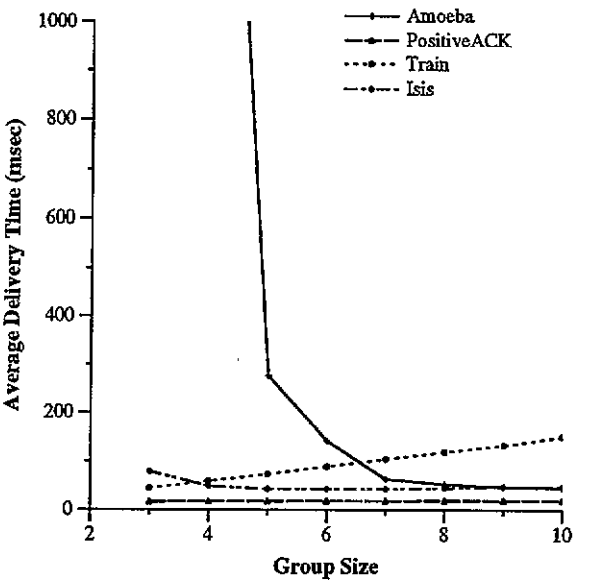

Figure 22. Comparison for $1/\lambda = 15.0$ ms; one message loss per broadcast.

protocols. For example, consider the scenario in which a process, say p_1 broadcasts an update u_1, and another process, say p_2 delivers u_1 and then broadcasts update u_2. If a causal order broadcast protocol is used, the sequencer is forced to order u_1 before u_2. No such restriction is present if a causal broadcast protocol is not used, as in the PA or the Amoeba protocols.

Finally, notice that the delivery times for the PA and Amoeba protocols are higher for lower mean interarrival times, because updates may be blocked for earlier broadcasts to complete when updates arrive at a faster rate. The delivery time for low mean update interarrival times is higher for Amoeba than for the PA protocol. The reason for this is that an update in Amoeba can be blocked at the sender if the sender is waiting for the completion of an earlier broadcast it originated, while in the PA protocol an update issued by a sender can be blocked at the sequencer only when a message carrying an earlier update from that sender to the sequencer is late. Thus, an update in the PA protocol is blocked only if the communication delay of the message carrying some earlier update, from the sender to the sequencer, is higher than the sum of the delay to the next update arrival and the communication delay of the message carrying this next update to the sequencer. On the other hand, an update in Amoeba is blocked at the sender only if the sum of the communication delay of the message carrying an earlier update from that sender to the sequencer and the communication delay of the message carrying the sequence number and this earlier update from the sequencer to the sender is more than the delay to the next update arrival. Clearly, chances of blocking of an update in Amoeba are higher than in the PA protocol.

The average delivery times of the protocols in the presence of one communication failure, are compared in figures 22 and 23 as a function of group size, and in figures 24 and 25 as a function of mean interarrival time. The first observation is that the protocols that are least affected by message losses are the PA and train protocols. Both use

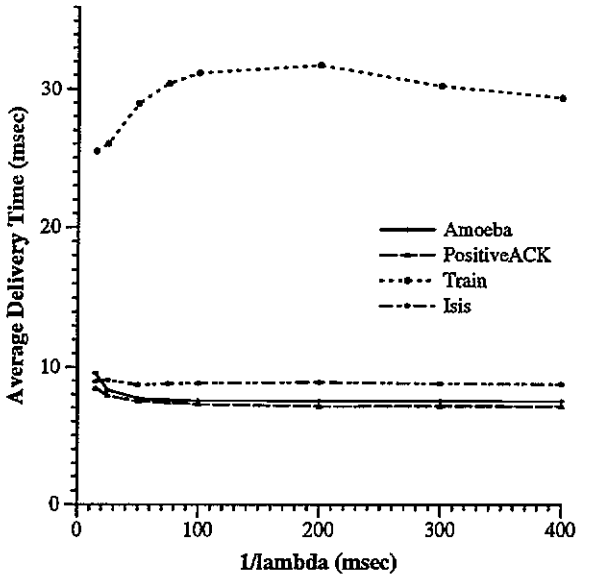

Figure 21. Comparison for group size 5; no message loss.

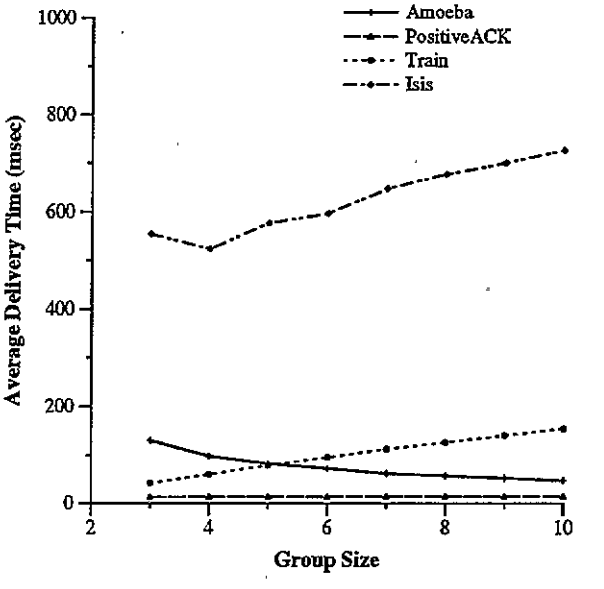

Figure 23. Comparison for $1/\lambda = 300.0$ ms; one message loss per broadcast.

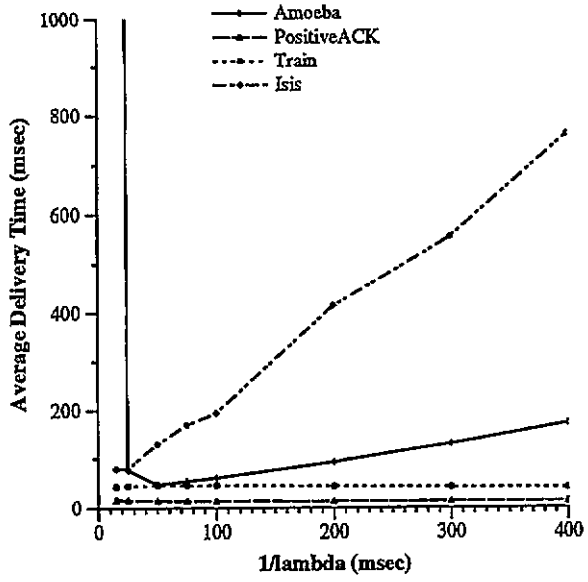

Figure 24. Comparison for group size 3; one message loss per broadcast.

timeouts to detect message losses, so the delivery times are increased by a fixed timeout interval when a message loss occurs. Since Isis and Amoeba use a negative acknowledgment strategy to detect message losses, their delivery times in the presence of a communication failure become dependent on the mean update interarrival time. As a result, the delivery times of these protocols are quite high (particularly for a small group size). These protocols are tuned for quick delivery in the absence of failures at the expense of poor performance in the presence of failures.

The effect of update blocking at the sender, as in Amoeba, versus the effect of update blocking at the sequencer, as in the PA protocol, becomes very clear now. Since, the delivery times are larger when messages may get lost, the time an update is blocked at the sender (in Amoeba) is much higher than the time an update is blocked at the sequencer (in the PA protocol). As a result, the delivery times in Amoeba are very high for low mean update interarrival times.

It is interesting to note that even though both Amoeba and Isis use negative acknowledgments, the average delivery time in Isis increases by a larger amount than in Amoeba when one communication failure occurs. The reason for this is that Isis uses negative acknowledgments at the lower FIFO layer while the Amoeba broadcast uses this technique at the higher broadcast layer. In a negative acknowledgment based protocol, a message loss is detected by a process q if q receives from p a message following the lost message from p. In the Isis case, process p can be any sending member: this will send the following message when the next update to be broadcast arrives from any of the group members. This, in effect, causes the message following the lost message to arrive earlier in the Amoeba protocol than in the Isis protocol, and contributes to the decrease in the average time to detect a message loss.

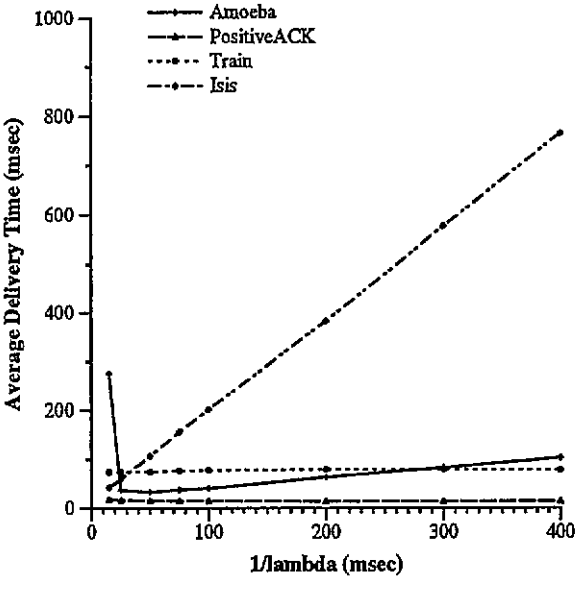

Figure 25. Comparison for group size 5; one message loss per broadcast.

5. Average stability time and maximum buffer size

5.1. Failure-free performance

5.1.1. The PA protocol. The average broadcast stability time is shown in figure 26 as a function of group size, and in figure 27 as a function of mean update interarrival times. Five delay components contribute to the PA stability time: (1) the message communication delay from the sender to the sequencer, (2) the time an update may wait at the sequencer, because the sequencer hasn't yet received some earlier update from the same sender, (3) the time needed to send the update from the sequencer to all group members and to receive back acknowledgments, (4) the time the sequencer has to wait until it receives some other update to broadcast, and (5) the time it takes for the new update to reach all group members. Only the last three delay components affect the stability time if the sender is the sequencer itself.

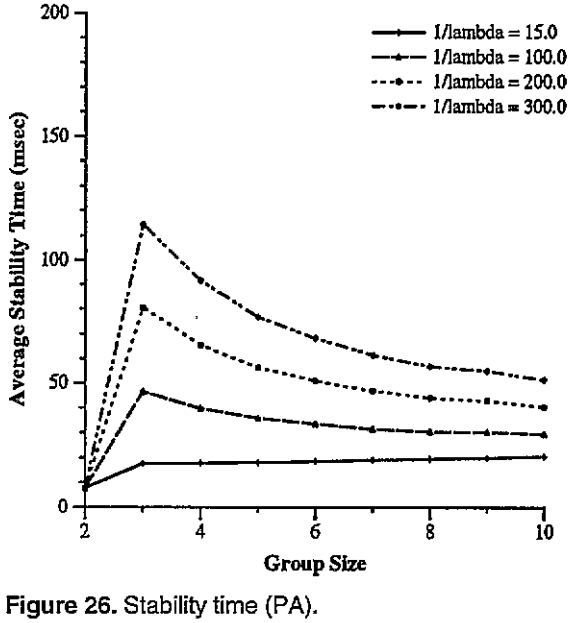

Figure 26. Stability time (PA).

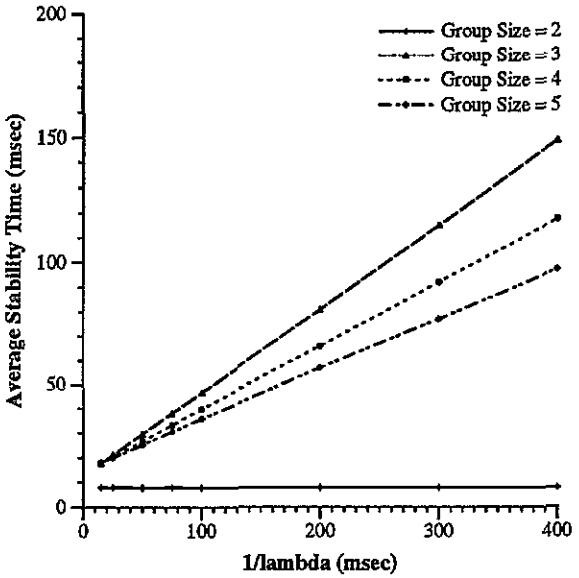

Figure 27. Stability time (PA).

There are three factors that affect average stability time when group size changes. First, the percentage of updates originated at the sequencer decreases when the group size increases, so an increase in group size causes the stability time to increase. Second, the third and the fifth delay components depend on the largest communication delays experienced by group members, and these increase with group size. Finally, the fourth delay component depends on the rate at which the sequencer sends messages to group members, which in turn depends on the per member update interarrival time. Since the per member update interarrival time decreases when the group size increases, this factor contributes to a decrease in average stability time for larger group sizes. As shown in figure 26, the third factor is dominant for when the mean update interarrival time is large: the stability time decreases when the group size increases (except for group size 2) for large mean update interarrival times. For the smallest mean update interarrival time considered (15.0 ms), the first two factors are dominant, and as result the stability time increases, albeit slightly, with the group size. Group size 2 is a special case because there is only one non-sequencer: as soon as the sequencer receives the acknowledgment from the non-sequencer, the broadcast is stable: the fourth and the fifth delay components do not contribute to the stability time in this case. As a result, the stability time for group size 2 is the smallest.

There are two factors that affect the average stability time as the mean update interarrival time changes. First, the number of messages arriving out of order is low for higher mean update interarrival times. This contributes to a decrease in the second stability time delay component when the mean update interarrival time increases. Second, the fourth delay component increases when the mean update interarrival time increases. As we see in figure 27, the second factor is dominant, and the stability time increases with the mean update interarrival time (except group size 2). Further, the rate of this increase is higher for lower group sizes (except for group size 2), because the fourth delay component depends on the per member update interarrival time which is smaller for larger group sizes. Group size 2 is a special case for the reasons mentioned earlier.

Finally, the maximum buffer size recorded for the simulation runs is shown in table 1 in the absence of failures. There are three main points to be noticed in these tables. First, the buffer size is larger for non-sequencers than for the sequencer, because the sequencer learns about stability of messages before non-sequencers. Second, buffer size increases with group size because there are more messages that are being broadcast. Third, buffer size decreases when the mean update interarrival time increases, because there are less messages being broadcast.

5.1.2. The Amoeba protocol. The average broadcast stability time is shown in figure 28 as a function of group size, and in figure 29 as a function of mean update interarrival time. The following delay components contribute to the stability time of an update in Amoeba: (1) the time an update must wait at the sender process if the sender is waiting for completion of an earlier broadcast, (2) the message communication delay from the sender to the sequencer, (3) the communication delay from the sequencer to all group members, (4) the time it takes for all group members to wait for their next update to broadcast and the time needed by these updates to arrive at the sequencer, and (5) the time needed by the sequencer to wait for a new update that follows the ones mentioned in (4) and the delay incurred by this new update in reaching all members. Only the last three delay components affect the stability time if the sender is the sequencer itself.

There are three factors that affect stability time when group size changes. These are similar to those mentioned for the PA protocol. First, as the group size increases, the percentage of updates broadcast by the sequencer decreases. Since the stability time of a sequencer update

Table 1. PositiveACK protocol: maximum buffer size; no failures.

PositiveACK		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	sequencer	7	4	3	3
	non-sequencer	6	4	3	3
Group size: 3	sequencer	9	6	5	3
	non-sequencer	15	7	6	4
Group size: 4	sequencer	10	7	5	4
	non-sequencer	16	8	6	5
Group size: 5	sequencer	12	8	6	4
	non-sequencer	18	9	8	5

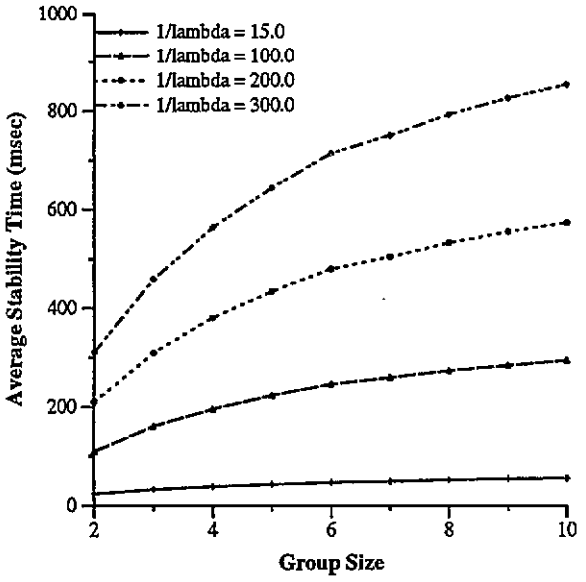

Figure 28. Stability time (Amoeba).

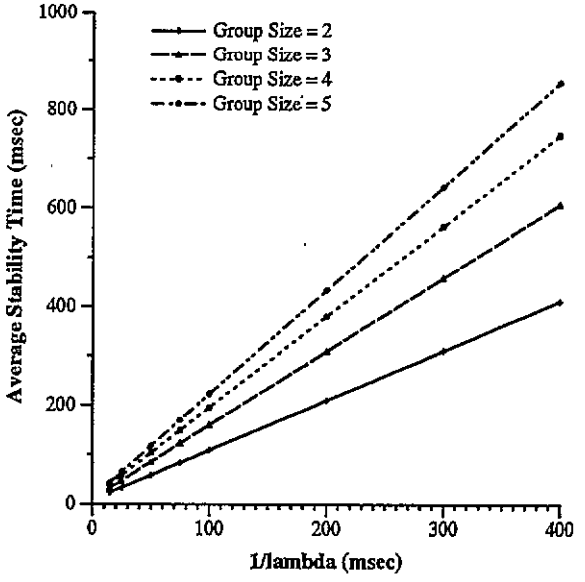

Figure 29. Stability time (Amoeba).

is less than that of a member update, the average stability time increases when the group size increases. Second, the communication delay from the sequencer to group members in the third and fifth component delays increases with group size. Third, the fourth delay component depends on the largest interarrival time experienced by all group members; this delay increases with group size. As we see in figure 28, the average stability time increases with group size because of all the above factors. The mean stability time is higher for larger mean update interarrival times because of the third factor. The average stability time increases when the mean interarrival time increases (figure 29) because of two reasons: the fourth and fifth component delays increase when the mean interarrival time increases.

Finally, the maximum message buffer size is shown in table 2 in the absence of failures. The variation in the maximum buffer sizes with group size or mean interarrival time is similar to that in the PA protocol, but because messages take longer to become stable, the buffer size requirements for Amoeba are greater than for the PA protocol (e.g. 152 maximum buffer space needed for a non-sequencer in Amoeba compared to 18 for the highest interarrival rate).

5.1.3. The train protocol. The average stability time is shown in figure 30 as a function of group size and in figure 31 as a function of mean update interarrival time. The delay components that contribute to the stability time of an update u are as follows: (1) the delay between the arrival of u and the arrival of the train, (2) the time the train takes to complete one round after u has been appended to it, (3) the delay to the arrival of a next update u' after u has been delivered by all group members, and (4) the time it takes for the train carrying u' to reach all group members. As shown in figure 30, the average stability time increases with the group size, because the first, second, and the fourth delay components increase with the group size. Furthermore, the rate of this increase is smaller for smaller group sizes and larger mean update interarrival times, because the third delay component (depending on the per member update interarrival time) is larger. The dependency between average stability time and mean update interarrival time is mostly caused by the third delay component. The average stability time increases when the mean update interarrival time increases because the per member update interarrival time increases. The rate of this increase is higher for smaller group sizes since the per member update interarrival time is higher for smaller group sizes.

The maximum message buffer sizes are shown in table 3 in the absence of failures. There are three points to be noted here: the maximum buffer size is larger for the trainmaster than non-trainmasters, it increases when the group size increases, and decreases when the mean interarrival time increases.

Table 2. Amoeba protocol: maximum buffer size; no failure.

Amoeba		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	sequencer	15	15	15	16
	non-sequencer	16	16	16	15
Group size: 3	sequencer	61	62	62	63
	non-sequencer	63	63	63	63
Group size: 4	sequencer	119	120	119	119
	non-sequencer	120	120	120	119
Group size: 5	sequencer	151	151	152	152
	non-sequencer	152	152	152	152

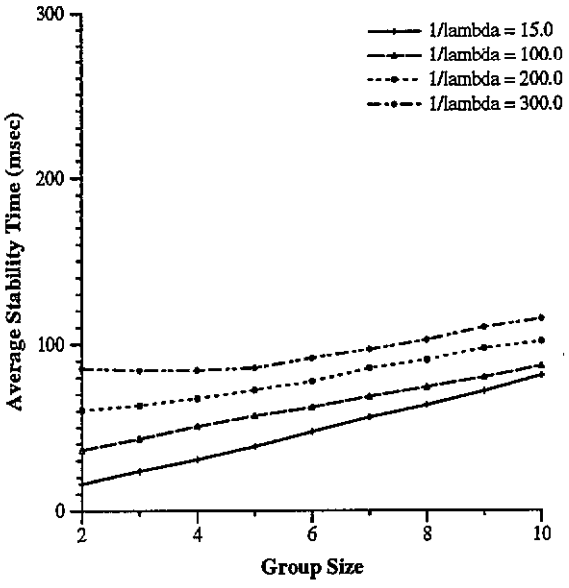

Figure 30. Stability time (train).

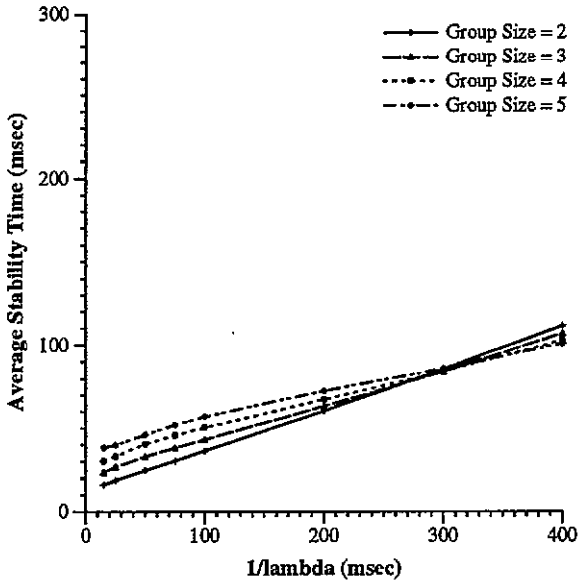

Figure 31. Stability time (train).

5.1.4. The Isis protocol. The average broadcast stability time for the Isis atomic broadcast is shown in figure 32 as a function of group size and in figure 33 as a function of mean update interarrival time. There are four delay components that contribute to the Isis stability time: (1) the communication delay to send a causal broadcast from the sender to all group members, (2) the communication delay to send the ordering information from the sequencer to all group members, (3) the wait time at the group members to broadcast their next update, and (4) the communication delay to send a causal broadcast containing these next updates to all group members. The stability time increases with group size because the second, third, and fourth delay components increase when the group size increases: the second and the fourth delay components depend on the largest communication delay experienced among all group members, and the third delay component depends on the largest update interarrival time experienced by group members. Furthermore, the rate of this increase is higher for lower group sizes because the percentage of broadcasts done by the sequencer is higher for lower group sizes, and the stability time for broadcasts originating at the sequencer is lower. The third delay component also depends on the mean update interarrival time, and so the stability time increases when the mean update interarrival time increases. The rate of this increase is higher for larger group sizes because, as explained earlier, the third delay component is larger for larger groups.

The maximum message buffer size needed by the Isis protocol in the absence of communication failures is shown in table 4. Buffer requirements are greater at the sequencer and they increase with group size.

5.2. Stability time and buffer requirements in the presence of a communication failure

5.2.1. The PA protocol. The average stability time in the presence of one communication failure per broadcast is shown in figure 34 as a function of group size, and in figure 35 as a function of mean update interarrival time. The loss of a message during a broadcast simply increases the average stability time by the timeout interval. The dependencies between the average stability time and the group size or the mean update interarrival time are similar to those observed in the failure free case. The maximum buffer size needed in the presence of one communication failures is shown in table 5. Once again, the dependencies here are similar to those in the failure-free case.

5.2.2. The Amoeba protocol. The average stability time in the presence of one communication failure is shown in figure 36 as a function of group size, and in figure 37 as a function of mean update interarrival time. Except for low

Table 3. Train protocol: maximum buffer size; no failures.

Train		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	trainmaster	13	11	8	6
	non-trainmaster	8	6	5	4
Group size: 3	trainmaster	24	14	12	6
	non-trainmaster	14	9	7	4
Group size: 4	trainmaster	28	18	13	8
	non-trainmaster	20	13	7	5
Group size: 5	trainmaster	44	23	16	8
	non-trainmaster	25	12	10	5

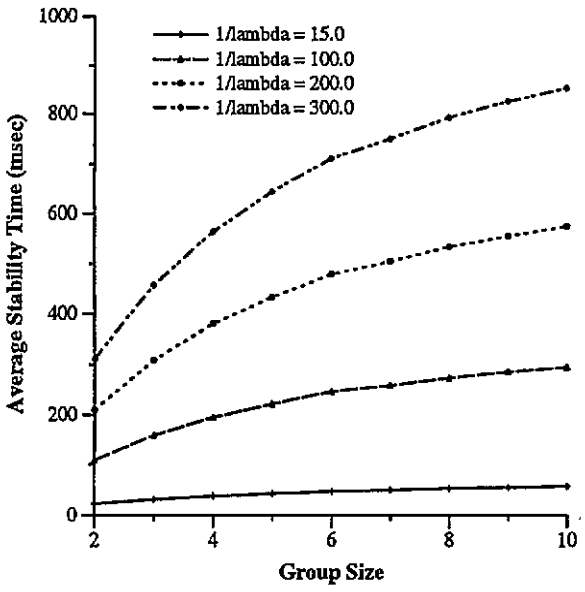

Figure 32. Stability time (Isis).

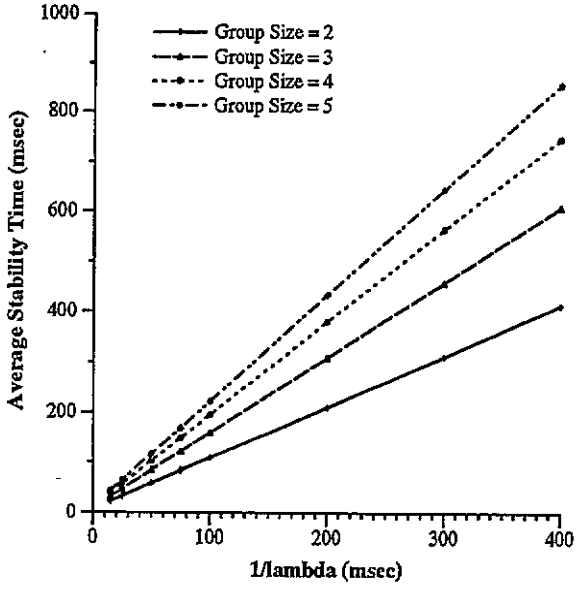

Figure 33. Stability time (Isis).

values of the mean update interarrival time (less than 50 ms), the dependencies between the average stability time, and the group size or the mean update interarrival time remain the same as in the absence of failures. The reason for the higher values of the average stability time at low mean interarrival times is that the first component of the stability delay in this case is very high and dominates the other delay components. The dependencies between the maximum message buffer size needed in the presence of one communication failure, and the group size or the mean interarrival time are similar to those in the failure-free case, as shown in table 6.

5.2.3. The train protocol. The average stability time in the presence of one communication failure per broadcast is shown in figure 38 as a function of group size, and in figure 39 as a function of mean update interarrival time. These dependencies are similar to those observed in the absence of communication failures. The dependencies between the maximum message buffer size needed in the presence of one communication failure, and the group size or the mean interarrival time are also similar to those in the failure-free case (table 7).

5.2.4. The Isis protocol. The average stability time in the presence of one communication failure per broadcast is shown in figure 40 as a function of group size and in figure 41 as a function of mean update interarrival time. Observe that the stability time graphs are similar to the delivery time graphs in the presence of failures given in figures 16 and 17. The reason for the similarity is that in both cases the time to detect a message loss dominates all the other delay components. The maximum message buffer sizes needed are shown in table 8.

Table 4. Isis ABCAST protocol: maximum buffer size; no failures.

Isis ABCAST		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	sequencer	44	44	44	41
	non-sequencer	15	11	8	7
Group size: 3	sequencer	126	125	127	127
	non-sequencer	125	125	125	125
Group size: 4	sequencer	313	309	305	303
	non-sequencer	273	271	266	265
Group size: 5	sequencer	410	417	412	412
	non-sequencer	285	283	278	278

Table 5. PositiveACK protocol: maximum buffer size; one failure.

PositiveACK		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	sequencer	10	6	5	3
	non-sequencer	9	5	4	3
Group size: 3	sequencer	14	8	6	4
	non-sequencer	17	9	7	5
Group size: 4	sequencer	18	8	7	4
	non-sequencer	18	10	8	5
Group size: 5	sequencer	18	9	7	4
	non-sequencer	20	11	8	6

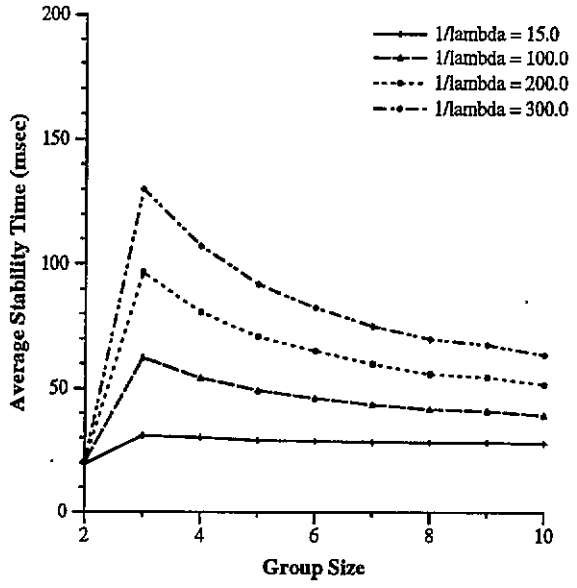

Figure 34. Stability time; 1 msg loss per broadcast (PA).

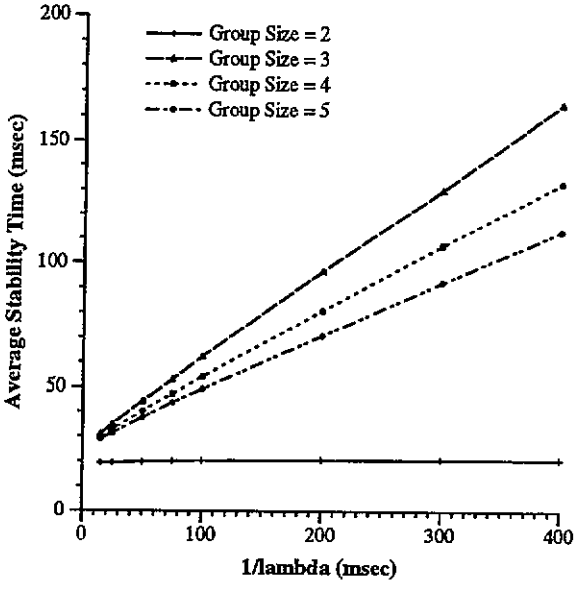

Figure 35. Stability time; 1 msg loss per broadcast (PA).

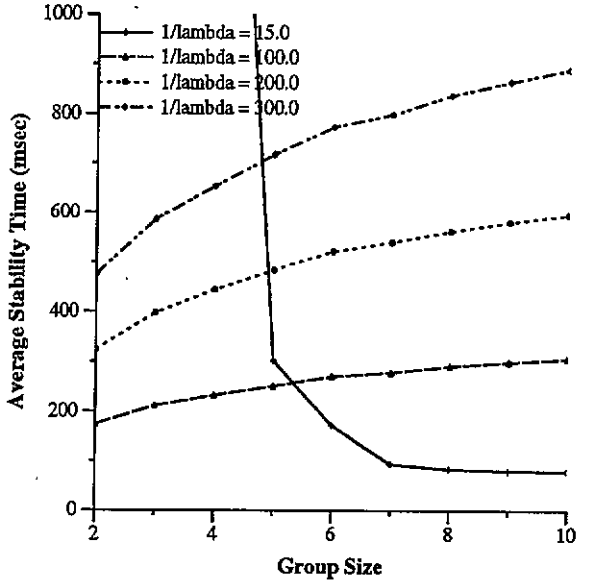

Figure 36. Stability time; 1 msg loss per broadcast (Amoeba).

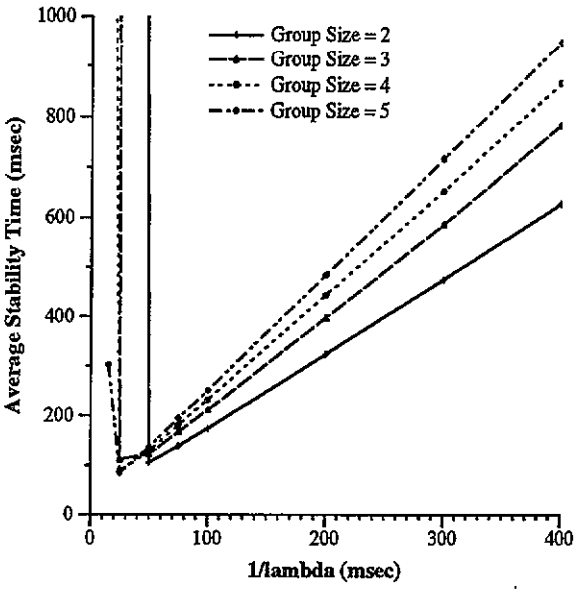

Figure 37. Stability time; 1 msg loss per broadcast (Amoeba).

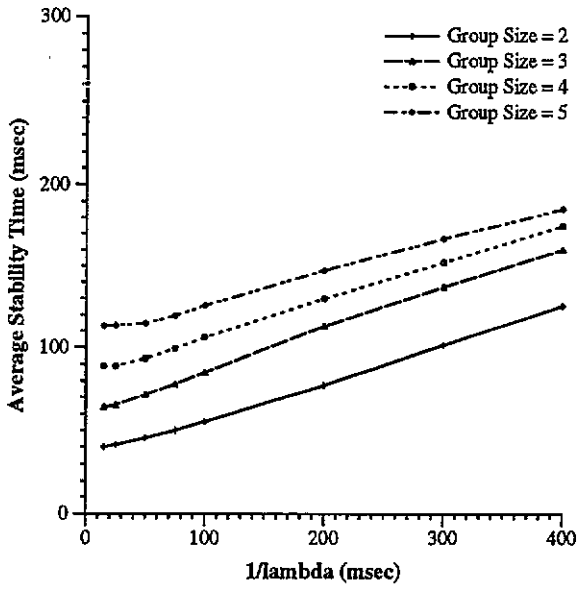

Figure 38. Stability time; 1 msg loss per broadcast (train).

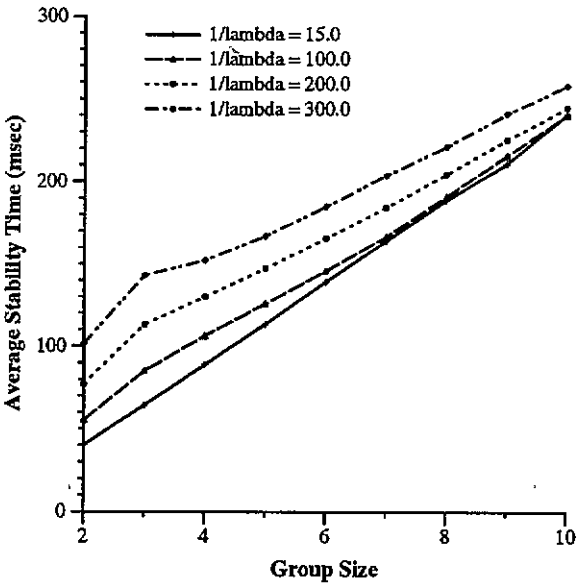

Figure 39. Stability time; 1 msg loss per broadcast (train).

Table 6. Amoeba protocol: maximum buffer size; one failure.

Amoeba		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	sequencer	94	21	17	17
	non-sequencer	4559	21	16	16
Group size: 3	sequencer	46	62	62	63
	non-sequencer	1090	63	63	63
Group size: 4	sequencer	130	121	119	119
	non-sequencer	368	121	120	119
Group size: 5	sequencer	152	152	154	152
	non-sequencer	165	152	154	152

Table 7. Train protocol: maximum buffer size; one failure.

Train		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 2	trainmaster	15	11	8	7
	non-trainmaster	11	7	6	4
Group size: 3	trainmaster	31	16	14	8
	non-trainmaster	18	9	8	4
Group size: 4	trainmaster	41	23	21	8
	non-trainmaster	25	14	10	6
Group size: 5	trainmaster	69	29	19	11
	non-trainmaster	42	17	12	6

Table 8. Isis ABCAST protocol: maximum buffer size; one failure.

Isis ABCAST		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Group size: 3	sequencer	190	135	131	144
	non-sequencer	192	187	189	208
Group size: 4	sequencer	323	311	322	306
	non-sequencer	280	271	335	267
Group size: 5	sequencer	412	434	416	420
	non-sequencer	292	385	279	278

5.3. Comparison

The failure-free average stability times of the protocols are compared in figures 42 and 43 as a function of group size, and in figures 44 and 45 as a function of mean update interarrival times. The stability times of all protocols are similar for low mean update interarrival times. For high mean update interarrival times the stability times of Isis and Amoeba (represented by the same line in figure 43) become high because of the use of negative acknowledgments. Since the only way a group member a learns that another member b has received an update u originating at a is to receive from b a message that follows receipt of u, the

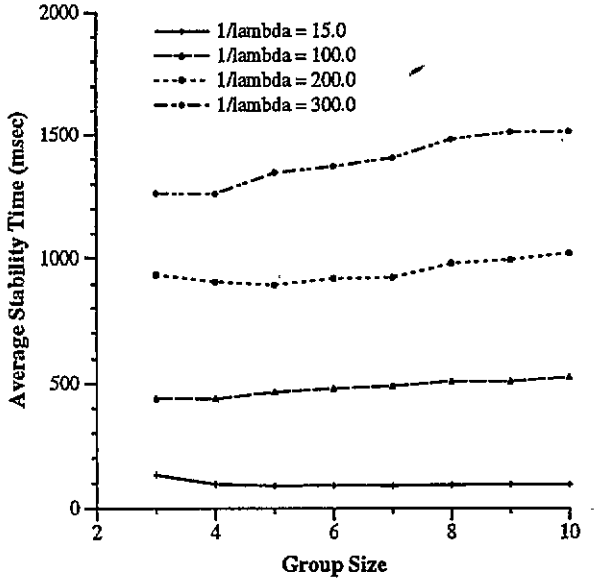

Figure 40. Stability time; 1 msg loss per broadcast (Isis).

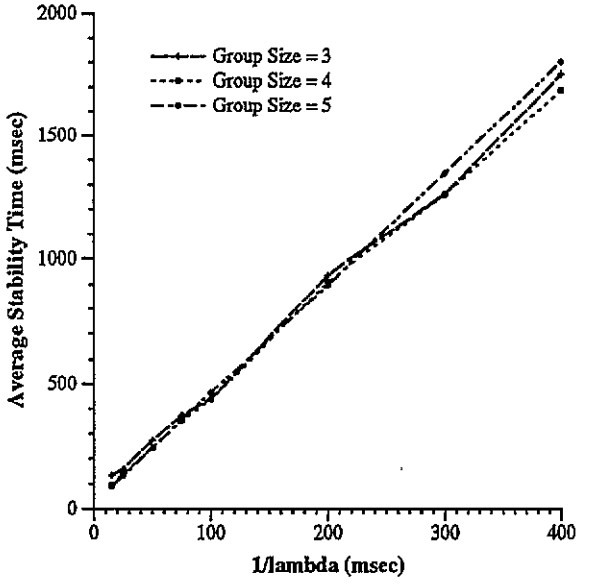

Figure 41. Stability time; 1 msg loss per broadcast (Isis).

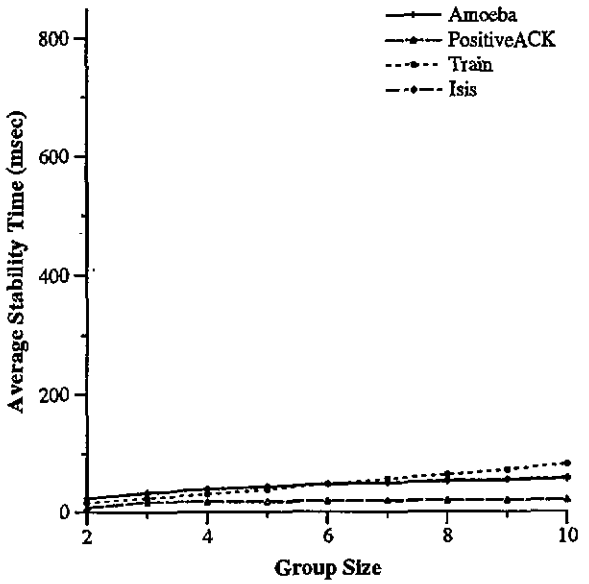

Figure 42. Comparison for $1/\lambda = 15.0$ ms; no message loss.

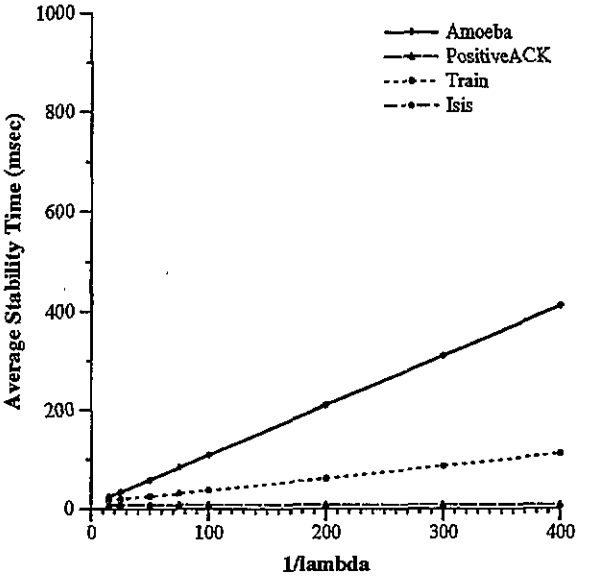

Figure 44. Comparison for group size 2; no message loss.

stability times in these protocols increase when the mean update interarrival time increases.

The stability time of the PA protocol is also dependent on the mean update interarrival time because the sequencer distributes the stability information in the 'following' message it sends to group members. However, the stability time in this case depends on the per member update interarrival time because the sequencer forwards every broadcast in the system. Hence, the stability time in the PA protocol is smaller than in Isis or Amoeba. The dependence of the stability time on the mean interarrival time is further reduced in the train protocol. Here, an update becomes stable when the train completes one more round after delivering the update at all group members. Thus, if any group member generates another broadcast request after the update has been delivered, the train continues its next round. Hence, the stability time in this case is dependent on a fraction of the per member update interarrival time. We see the effect of this in figure 45, where the stability in the train protocol gets better and better (compared to the other protocols) as the mean interarrival time increases.

The average stability times in the presence of one communication failure are plotted in figures 46 and 47 as a function of group size, and in figures 48 and 49 as a function of mean update interarrival time. The stability times in Isis and Amoeba are very high (for large mean interarrival times) compared to those in the PA and train protocols, because of the dependence on the mean update interarrival times. In the train and PA protocols, the dependency between stability time and mean update interarrival time is minimized, so these protocols achieve good stability times. The high stability time at low mean interarrival times in Amoeba is caused by update blocking at the sender, as explained earlier.

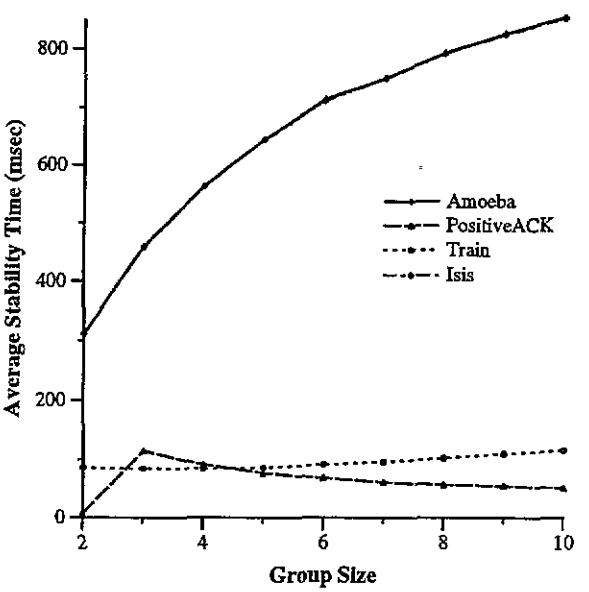

Figure 43. Comparison for $1/\lambda = 300.0$ ms; no message loss.

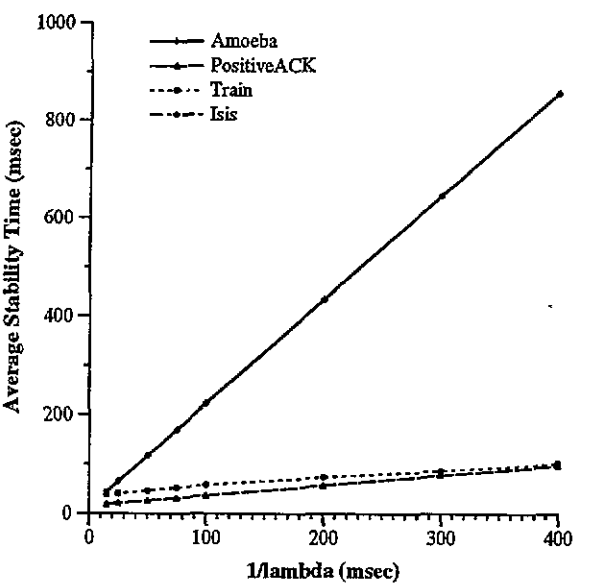

Figure 45. Comparison for group size 5; no message loss.

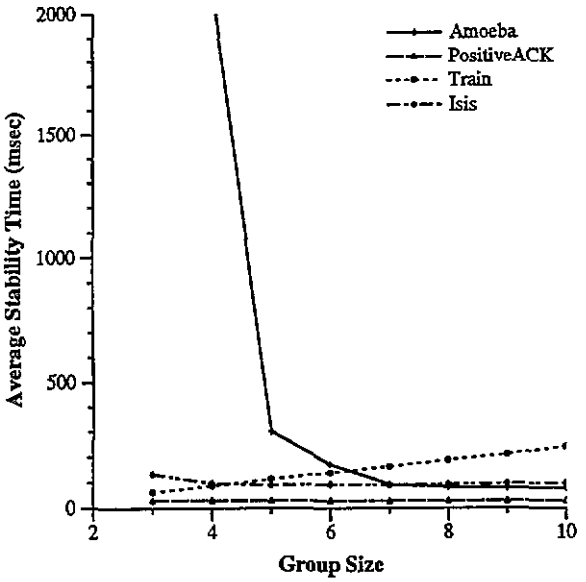

Figure 46. Comparison for $1/\lambda = 15.0$ ms; one message loss per broadcast.

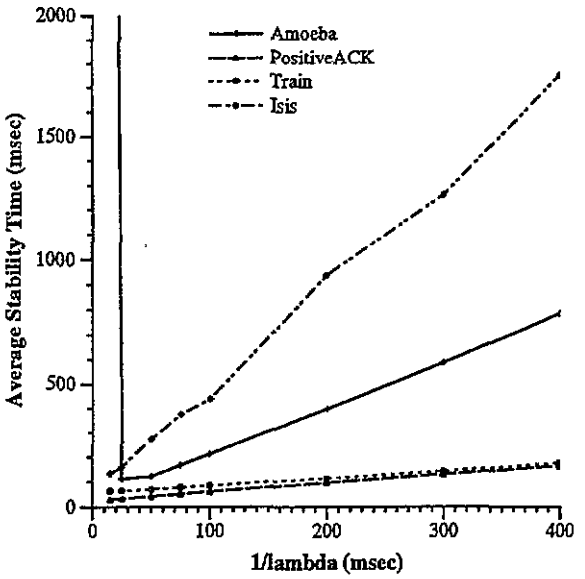

Figure 48. Comparison for group size 2; one message loss per broadcast.

5.4. Maximum buffer size

Failure-free maximum buffer sizes for groups with 3 and 5 members are compared in tables 9 and 10, respectively. Tables 11 and 12 give the maximum buffer sizes in the presence of one communication failure per broadcast. The maximum buffer sizes are significantly smaller for the train and PA protocols, both in the absence and in the presence of communication failures. The explanation for this is the same as that given earlier for the low stability times of these protocols compared to Isis or Amoeba. The effect of a message loss on buffer size requirement is quite significant in the Isis and Amoeba negative acknowledgment based protocols, while its effect on positive acknowledgment based protocols such as the train and PA is minimal.

6. Average number of messages per broadcast

6.1. The PA protocol

The average number of messages sent per broadcast as a function of mean update interarrival time is shown in figure 50 in the absence of any failures, and in figure 51 in the presence of one communication failure per broadcast. Although we could have shown these values in tables, we have chosen to present them in graphs to be consistent with the presentation for the other protocols for which graphs are needed. In the absence of failures, the average number of messages sent per broadcast is independent of the mean interarrival time and increases linearly with the group size. The dependency between the number of messages sent per broadcast and the mean update

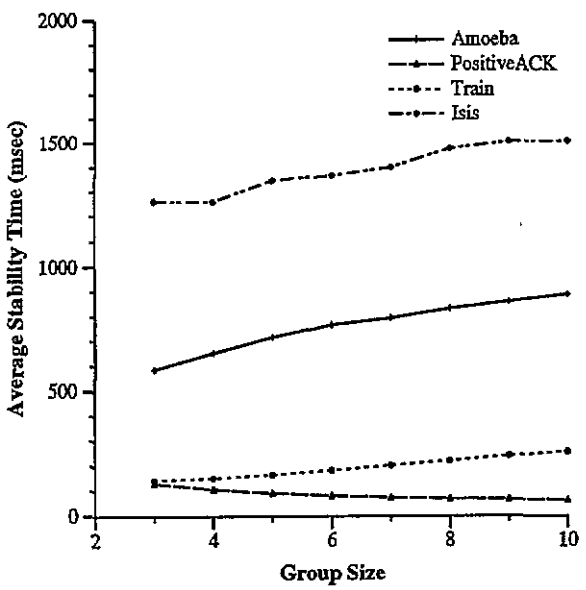

Figure 47. Comparison for $1/\lambda = 300.0$ ms; one message loss per broadcast.

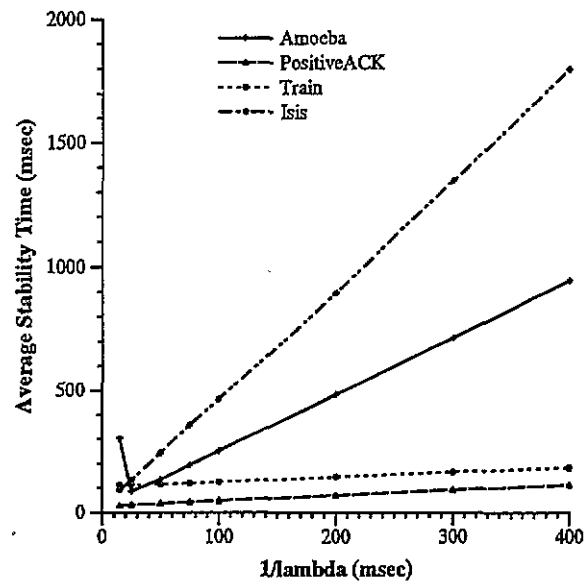

Figure 49. Comparison for group size 5; one message loss per broadcast.

Table 9. Maximum buffer size: group size 3, no failures.

Group Size: 3		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Amoeba	sequencer	61	62	62	63
	non-sequencer	63	63	63	63
PositiveACK	sequencer	9	6	5	3
	non-sequencer	15	7	6	4
Train	trainmaster	24	14	12	6
	non-trainsmaster	14	9	7	4
Iris ABCAST	non-sequencer	125	125	125	125

Table 10. Maximum buffer size: group size 5; no failures.

Group Size: 5		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Amoeba	sequencer	151	151	152	152
	non-sequencer	152	152	152	152
PositiveACK	sequencer	12	8	6	4
	non-sequencer	18	9	8	5
Train	trainmaster	44	23	16	8
	non-trainmaster	25	12	10	5
Isis ABCAST	sequencer	410	417	412	412
	non-sequencer	285	283	278	278

Table 11. Maximum buffer size: group size 3; one failure.

Group Size: 3		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Amoeba	sequencer	46	62	62	63
	non-sequencer	1090	63	63	63
PositiveACK	sequencer	14	8	6	4
	non-sequencer	17	9	7	5
Train	trainmaster	31	16	14	8
	non-trainmaster	18	9	8	4
Isis ABCAST	sequencer	190	135	131	144
	non-sequencer	192	187	189	208

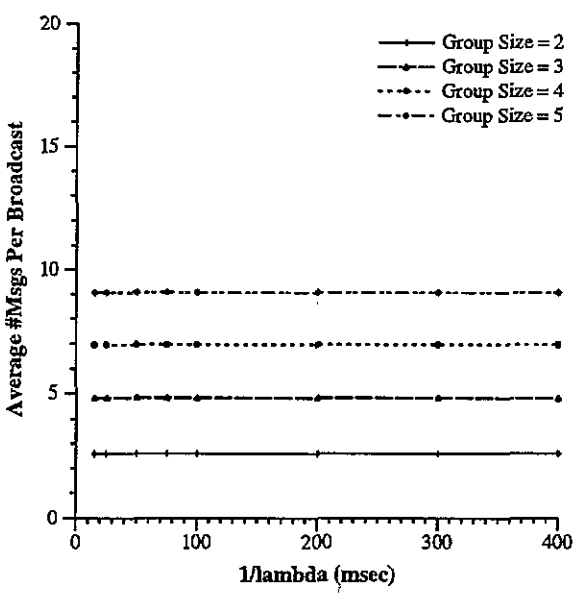

Figure 50. Number of messages pre broadcast (PA).

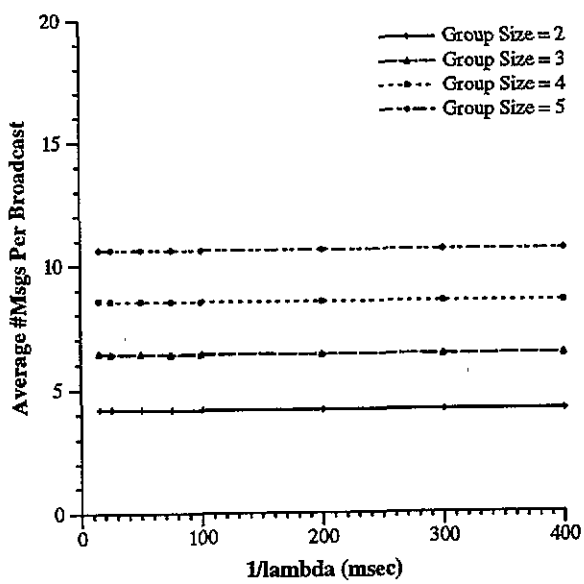

Figure 51. Number of messages per broadcast, 1 msg loss per broadcast (PA).

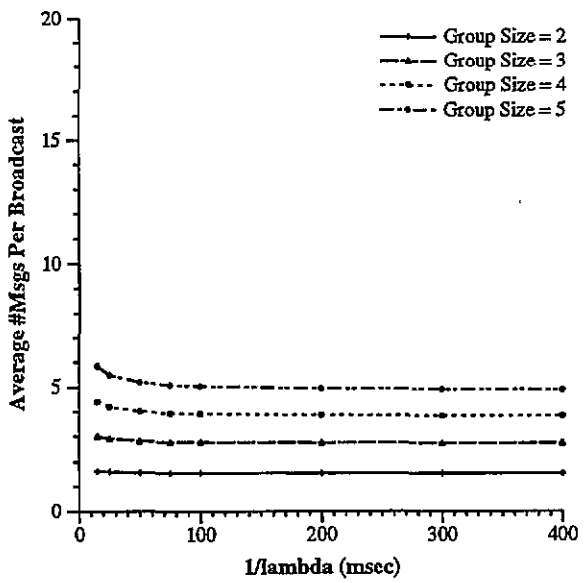

Figure 52. Number of messages pre broadcast (Amoeba).

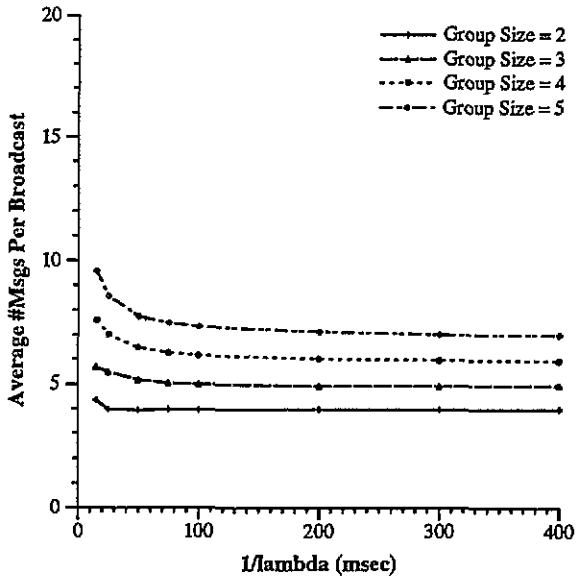

Figure 53. Number of messages per broadcast; 1 msg loss per broadcast (Amoeba).

interarrival time in the presence of a single communication failure per broadcast is similar to that in the failure free case, with the only difference that the number of messages sent per broadcast is higher.

6.2. The Amoeba protocol

The average number of messages sent per broadcast in the absence of failures is shown in figure 52 as a function of mean update interarrival times. Figure 53 displays the average number of messages per broadcast in the presence of one communication failure. The messages counted include the first message sent from a sender to the sequencer, the messages sent from the sequencer to all group members, as well as the retransmit requests from group members to the sequencer if messages arrive out of order. For larger values of the mean update interarrival time (greater than 50 ms), the average number of messages sent per broadcast increases linearly with the group size, and is independent of the mean update interarrival time. The increase with the group size is explained by the fact that the number of messages sent from the sequencer to group members increases with group size. The reason for the higher number of messages observed at low mean interarrival times is that the likelihood of messages arriving out of order increases when the update interarrival time decreases.

6.3. The train protocol

The average number of messages sent per broadcast in the absence of failures is shown in figure 54 as a function of mean update interarrival time. This number increases with the mean update interarrival time, and approaches *group size* + 2. Indeed, when the updates arrive slowly, the train is mostly idle, so the messages used to broadcast an update at large interarrival times consist of the messages to request and to transfer the train and the messages to circulate the train among group members. On the other hand, when the mean update interarrival time is small, the train carries more than one update and is never idle, so that no messages to request and transfer the train are needed. As a result, the average number of messages per broadcast decreases when the mean update interarrival times decreases. This dependency remains the same in the presence of one communication failure per broadcast, with the difference that the average number of messages sent per broadcast is larger see figure 55.

6.4. The Isis protocol

The dependency between the average number of messages sent per broadcast and the mean update interarrival time is shown in figure 56 in the absence of failure, and in figure 57 in the presence of one communication failure per broadcast. The messages include the messages sent from a sender

Table 12. Maximum buffer size: group size 5; one failure.

Group Size: 5		$1/\lambda = 15.0$	$1/\lambda = 50.0$	$1/\lambda = 100.0$	$1/\lambda = 400.0$
Amoeba	sequencer	152	152	154	152
	non-sequencer	165	152	154	152
PositiveACK	sequencer	18	9	7	4
	non-sequencer	20	11	8	6
Train	*trainmaster*	69	29	19	11
	non-trainmaster	42	17	12	6
Isis ABCAST	sequencer	412	434	416	420
	non-sequencer	292	385	279	278

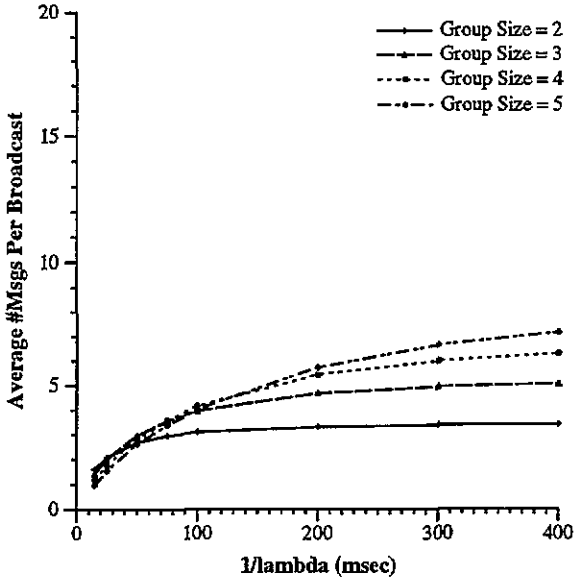

Figure 54. Number of messages per broadcast (train).

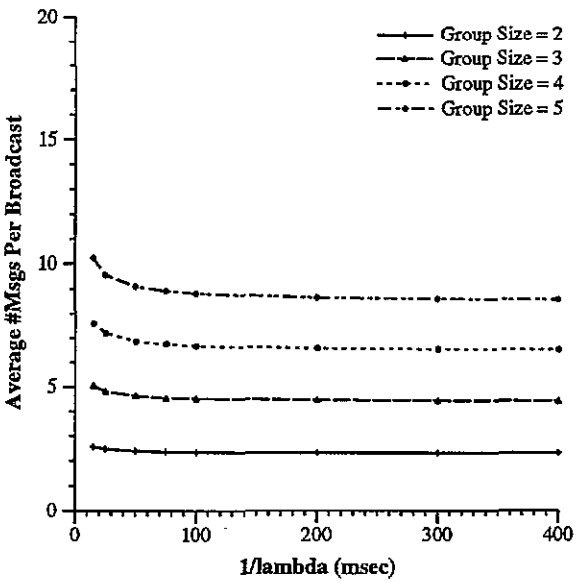

Figure 56. Number of messages per broadcast (Isis).

to all group members, the ordering messages sent from the sequencer to all group members, and the retransmit messages sent at the FIFO layer in case some message is lost or arrives out of order. The average number of messages sent per broadcast is independent of the mean interarrival time for $1/\lambda > 25.0$, and varies linearly with the group size. When the mean update interarrival time is small, the probability of messages arriving out of order is high at the FIFO layer. This causes additional retransmit request messages to be sent and thus increases the average number of messages sent per broadcast for lower values of the mean update interarrival time.

6.5. Comparison

The average number of messages per broadcast in the absence of failures is plotted in figures 58 and 59 as a function of mean interarrival time. The first point to notice is that for all protocols, the average number of messages per broadcast reaches a stable value when the mean interarrival time increases. Also, the two most interesting protocols when it comes to message density are the train and Amoeba protocols. Figure 58 shows that for group size 3, the train protocol has the lowest message density for interarrival times smaller than 50 ms, and that the Amoeba protocol has the lowest message density for interarrival times greater than 50 ms. For group size 5, the train protocol is better when the mean interarrival time is less than 175.0 ms, while the Amoeba protocol is better for interarrival times higher than 175 ms. Another point to note is that the average number of messages per broadcast increases for low mean interarrival times in all protocols except the train protocol. This is a direct result of the concurrency built in the PA, Amoeba and Isis protocols: at low mean interarrival times, there are more messages

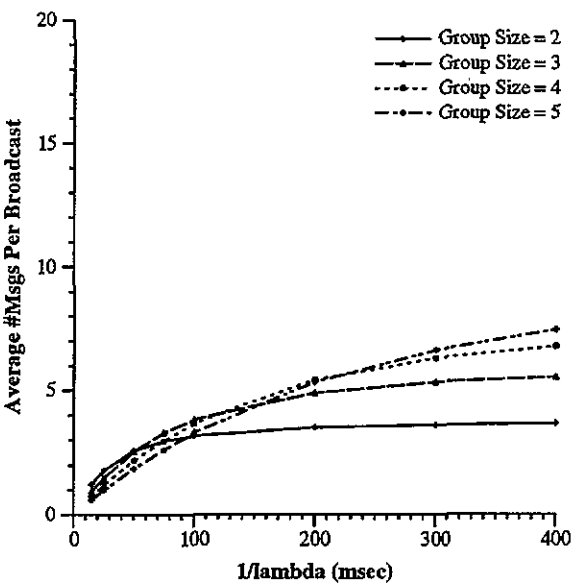

Figure 55. Number of messages per broadcast; 1 msg loss per broadcast (train).

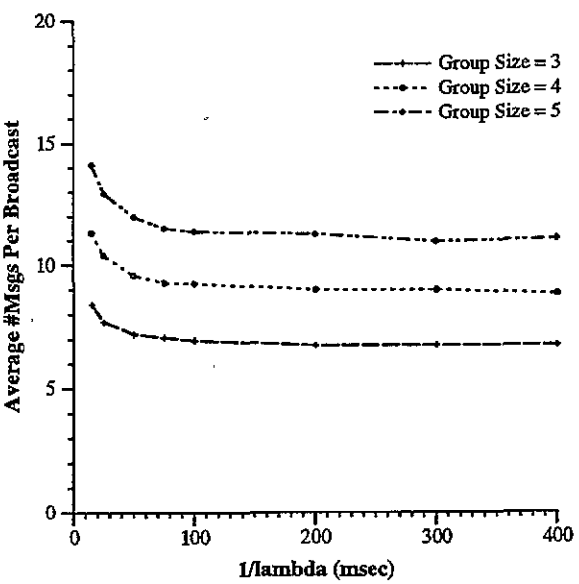

Figure 57. Number of messages per broadcast; 1 msg loss per broadcast (Isis).

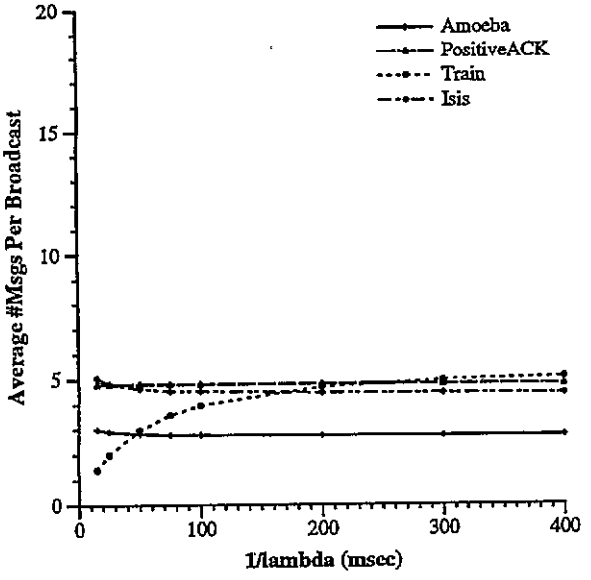

Figure 58. Comparison for group size 3; no message loss.

arriving out of order for sequencer based protocols and as a result there are more retransmit requests.

The average number of messages per broadcast in the presence of one communication failure per broadcast is plotted in figures 60 and 61 as a function of the mean interarrival time. As in the failure-free case, the two protocols that dominate are the train and Amoeba: for group size 3 the train is better for interarrival times smaller than 200 ms and Amoeba is better for higher values; for group size 5 the train is best for interarrival times less than 350.0 ms and Amoeba becomes best for values beyond 350 ms.

7. Distribution of processing load

The distribution of processing load refers to how busy a group member is in relation to other group members over some sufficiently long period of time in which all group

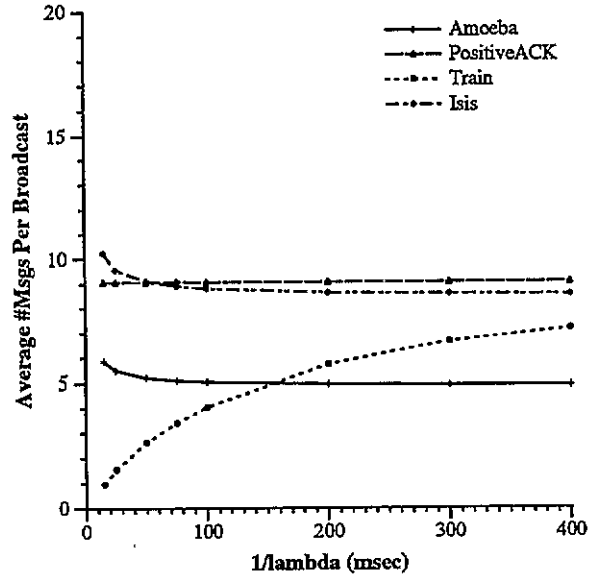

Fig. 59. Comparison for group size 5; no message loss.

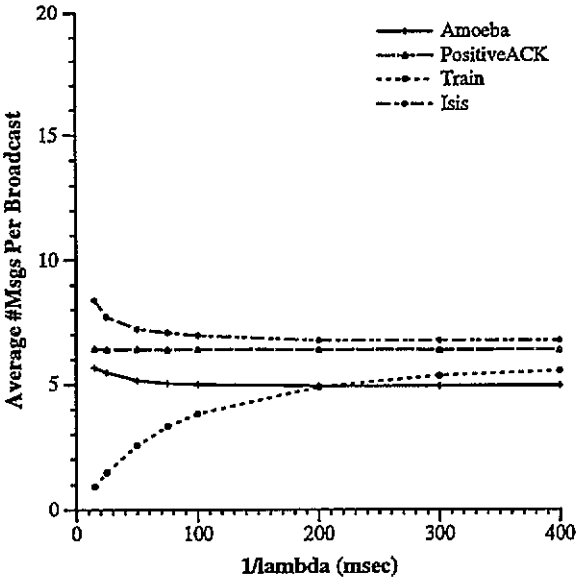

Figure 60. Comparison for group size 3; one message loss per broadcast.

members initiate about the same number of broadcasts. We estimate this based on the number of messages processed (i.e. the sum of the number of messages sent and the number of messages received) by each group member over this period.

In the PA, Amoeba, and Isis protocols, the number of messages processed by non-sequencer group members were observed to be approximately equal. Similarly, in the train protocol, the number of messages processed by members other than the trainmaster were observed to be approximately equal. In tables 13 and 14, the number of messages processed by the sequencer and non-sequencers as a percentage of the total number of messages processed is shown.

The following observations can be made. In the sequencer-based protocols, the percentage of the number of messages processed by the sequencer far exceeds those

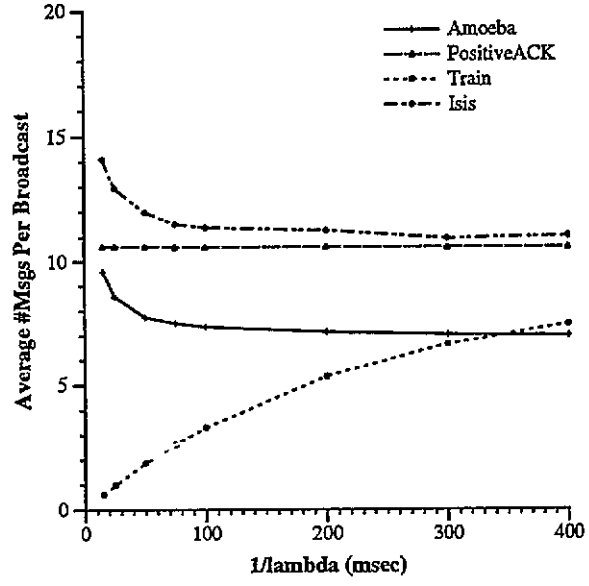

Fig. 61. Comparison for group size 5; one message loss per broadcast.

Table 13. Percentage of the number of messages processed (group size = 3).

Group Size: 3		No failure		Failure	
		$1/\lambda = 15.0$	$1/\lambda = 300.0$	$1/\lambda = 15.0$	$1/\lambda = 300.0$
Amoeba	sequencer	50.04	50.00	54.45	54.08
	non-sequencer	24.98	25.00	22.77	22.96
PositiveACK	sequencer	50.00	50.00	50.16	50.15
	non-sequencer	25.00	25.00	24.92	24.92
Train	trainmaster	33.34	33.34	33.59	33.37
	non-trainmaster	33.33	33.33	33.21	33.32
Isis ABCAST	sequencer	41.68	41.67	44.62	44.62
	non-sequencer	29.16	29.16	27.69	27.69

processed by non-sequencers, while in the train protocol all group members process approximately the same number of messages. Thus the processing load is equally distributed among all group members in the train protocol, while the sequencer is overloaded in the PA, Amoeba, and Isis protocols. Second, the difference between the percentage of the number of messages processed by the sequencer and the other group members in the PA, Amoeba, and Isis protocols increases with the increase in group size. Third, this difference is smaller in the Isis protocol than in the PA and Amoeba protocols. The reason for this is that in the PA and Amoeba protocols, the sequencer is responsible for recovery from communication failures (it responds to all retransmission requests in the Amoeba protocol, and retransmits messages in the PA protocol if a positive acknowledgment is not received from some group member), while in the Isis protocol some of the retransmission requests are handled by update senders.

8. Some design hints for good performance

So far we have studied the performance of the PA, Amoeba, train and Isis protocols relative to each other. In this section, we will identify some features that are useful in designing broadcast protocols that perform well.

All broadcast protocols that we investigated enforce a sender based ordering on top of the total ordering that they provide: if a sender broadcasts an update u_1 followed by an update u_2, then these protocols ensure that group members deliver u_1 before u_2. The techniques used to ensure sender ordering are different for different protocols. In the PA protocol, a sender attaches a sequence number to every update it originates and the sequencer orders the updates in a manner consistent with the sender-generated ordering. In the Amoeba protocol, a sender issues only one update at a time. In Isis, sender-ordering is achieved by use of the underlying FIFO channel between a member and the sequencer. Finally in the train protocol, each sender appends the updates to the train in the order it has received them. For large mean interarrival times, all these techniques have similar effects on the performance of the protocol. However, for low mean interarrival times, the technique of update blocking at the sender, as used in Amoeba, increases the delivery and stability times significantly. This increase is even more significant in the presence of message losses. Hence, if good performance in the presence of high update arrival rates is desired, blocking at the sender should be avoided. However, this technique does reduce protocol complexity as well as the load on the sequencer, so it is acceptable if it is anticipated that updates will arrive spaced by long delays and bursty arrivals are unlikely. The Isis broadcast protocol imposes additional causal restrictions on the delivery of broadcast updates. As we have seen, this contributes to increasing the delivery and stability times. Thus, such a restriction is a useful feature only if applications really need causality to be preserved.

The delivery times in the absence of failures are independent of the mean update interarrival time for sequencer based protocols, while the train protocol performs in general better at high update arrival rates. The stability and message loss detection times for negative acknowledgment based protocols, such as Amoeba and Isis, are dependent on the mean update interarrival time. This dependency leads to large delivery times in the

Table 14. Percentage of the number of messages processed (group size = 5).

Group Size: 5		No failure		Failure	
		$1/\lambda = 15.0$	$1/\lambda = 300.0$	$1/\lambda = 15.0$	$1/\lambda = 300.0$
Amoeba	sequencer	51.09	51.02	53.13	52.79
	non-sequencer	12.23	12.24	11.72	11.80
PositiveACK	sequencer	50.00	50.00	50.27	50.24
	non-sequencer	12.25	12.25	12.43	12.44
Train	trainmaster	20.02	20.02	20.24	20.19
	non-trainmaster	19.97	19.97	19.93	19.94
Isis ABCAST	sequencer	36.21	35.87	41.44	40.32
	non-sequencer	15.95	16.03	14.63	14.92

presence of communication failures and to larger stability times, compared to the train and PA protocols, both in the absence and in the presence of communication failures. Furthermore, larger stability times also increase the buffer size requirements. Thus, if the goal is decent performance at low update arrival rates or when failures occur, the dependency of the stability and message loss detection times on the mean update interarrival time should be minimized as much as possible.

9. Conclusion

Choosing an appropriate atomic broadcast protocol for a fault-tolerant application based on data replication is a delicate task. This paper provides a contribution towards solving this problem by studying the comparative performance of sequencer-based and train protocols under identical conditions. This is, to our knowledge, the first comparative study ever attempted into the performance of asynchronous atomic broadcast protocols. Since such protocols are expected to become more important in the future, given the ever increasing availability requirements placed on distributed systems, we expect this to be a first step towards a comprehensive understanding of the performance characteristics of group broadcast protocols.

Our decision to investigate the comparative performance by simulation rather than by a full-fledged implementation enabled us to get results faster and provided us with a fair comparison between protocols, since their performance was investigated by using identical assumptions and simulation conditions. In addition to this, our choice of investigation method proved useful in many other ways. First, it forced us to understand the details of the protocols in order to simulate their behaviour. Second, in many instances, the protocol behaviour observed from simulations was difficult to anticipate from the published protocol descriptions because of inherent complexity. For example, the more than linear decrease in the number of messages sent per broadcast by the train protocol when the update arrival rate increases linearly was a surprise for us; the effect of update blocking at the sender in Amoeba at high update arrival rates was more significant than we could foresee; the better delivery and stability times of Amoeba compared to Isis, even though both of these protocols use similar negative acknowledgment techniques, was unexpected—we provided an explanation that, we believe, is reasonable, but without our simulation work we couldn't even think about such differences in behaviour; the unsatisfactory performance of the Tandem protocol was a surprise, and this forced us to investigate the positive acknowledgment variant that, to the best of our knowledge, is a new protocol with interesting performance characteristics. Third, although the effect of the update arrival rates or the group size on different performance indexes could, in general, be predicted in many cases, the simulation provided us with interesting actual values of the mean update interarrival time and the group size for which these protocols could perform best. For example, we have observed the existence of a value for the mean update interarrival time (25 ms) for which the effect of messages arriving out of order on the delivery times, in the absence of failures, in the PA protocol becomes insignificant (figure 3). Similarly, we have observed that the consequences on the average stability time of sender blocking due to message losses and high arrival rates in Amoeba become insignificant beyond a certain group size (7) when compared with Isis (figure 4*b*). We have also shown the existence of a mean update interarrival time value such that for lower interarrival times the train protocol sends the minimum number of messages per broadcast while for higher values the Amoeba protocol sends the minimum number of messages per broadcast. For example, for a group of size 5, the train protocol sends less messages per broadcast when the mean update interarrival time is smaller than 150 ms while Amoeba sends less messages per broadcast for values higher than 150 ms (figure 5*b*).

While there is no single protocol with the best overall performance under all assumed parameter values, we have attempted to identify their relative strengths and weaknesses. In as far as protocol complexity is concerned, our opinion is that the simplest to understand and implement is the train protocol, followed by, in order of increasing complexity: the Positive Acknowledgment, Amoeba and Isis protocols. The best delivery times in the absence of failures are provided by sequencer-based protocols such as the Positive Acknowledgment, Amoeba and Isis. While the Positive Acknowledgment protocol has a delivery time only slightly worse than Amoeba and Isis in the absence of failures, it has the best performance in the presence of communication failures. When communication failures occur, the delivery times of negative acknowledgment based protocols, such as Amoeba and Isis, grow to very large values, larger for Isis than for Amoeba because of the use of negative acknowledgments at the lower FIFO transport layer. The stability times for low mean update interarrival times are similar for all protocols. When the mean update interarrival time increases, the stability times of Amoeba and Isis become very large. We have explained this by describing the strong dependency which exists between stability time and mean interarrival time for such negative acknowledgment based protocols. As far as message density is concerned, the best protocols are the train and Amoeba. The train protocol has the lowest number of messages per broadcast for high update arrival rates and Amoeba has the lowest number of messages per broadcast for low update arrival rates, both when no failure occur and when a communication failure occurs. As a general comment, the train protocol performs better when the update interarrival rate increases, while the other protocols perform better when the update interarrival rate decreases. Negative acknowledgment based protocols such as Amoeba and Isis require more buffer storage than protocols based on timeouts such as the PA and the train. Finally, the train protocol distributes processing load evenly among group members, while the sequencer based protocols overload the sequencer. The overloading increases when the group size increases.

While our study was done for point-to-point networks,

it is worth observing that both sequencer-based and train-based protocols allow optimizations when the underlying communication network is a broadcast channel instead of a point-to-point network. The sequencer based protocols would then use the channel multicast capability each time a member must send information that must reach all group members. A train based protocol optimized for a broadcast channel would dump all updates accumulated at a sender on the channel when the train (or token in this case) arrives at a sender instead of appending them to the train as in the point-to-point case. Instead of containing updates like in the point-to-point case, the train (or token) would carry only update reception, delivery and stability information that would be constantly updated by each visited member. Such information would be needed to enable each member to detect lost updates and ask for retransmissions and to learn which received updates have become stable. We expect such optimizations to share many of the performance characteristics mentioned, while differing on some important other points.

Another point to be noted is that our study was done under the assumption that communication has omission performance failure semantics and out of order messages can occur because of retries or use of different routes between message sources and destinations. All the protocols investigated were designed with this assumption in mind. While this is true for most contemporary communication service technology, this is not true for newer network technologies such as ATM, that preserve source-imposed message orderings and reduce the likelihood of message losses caused by insufficient buffer space allocation along the routes followed by messages. Such changes in failure semantics will affect the relative performance of the atomic broadcast protocols that will run on ATM communication services. It would not be surprising if new protocols that exploit the new failure semantics provided by such new technologies emerged.

Finally, our goal was to get a general feeling of the strengths and weaknesses of the investigated atomic broadcast protocols. As a result, we chose not to consider many useful optimizations that can be applied to one or more of these protocols in some specific computing environment. Clearly, these optimizations will affect to some degree the performance indexes we have measured but we are not sure if it will fundamentally change the observed performance trends. We expect that the issue of the study of the relative performance of atomic broadcast protocols for various other communication environments such as broadcast channels or ATM, with optimizations appropriate for each such environment, will become an exciting and active research area in the future.

Acknowledgments

This work was supported by grants from the Powell Foundation, the Airforce Office of Scientific Research, Sun Microsystems and the Microelectronics Innovation and Computer Research Opportunities of California. RdB, visiting from the University of Nijmegen, The Netherlands, was supported by grants from the Foundation of Renswoude at the Hague and the Dutch Ministry of Education and Sciences.

References

[1] Birman K, Schiper A and Stephenson P 1991 Lightweight causal and atomic group multicast *ACM Trans. Comput. Syst.* **9** 272–314

[2] Carr R 1985 The Tandem global update protocol *Tandem Syst. Rev.* June

[3] Cristian F 1991 Asynchronous atomic broadcast *IBM Tech. Disclosure Bull.* **33** (9) 115–6, also presented at the *1st IEEE Workshop on Management of Replicated Data (Houston, TX, 1990)*

[4] Cristian F, Aghili H, Strong R and Dolev D 1985 Atomic broadcast: from simple message diffusion to Byzantine agreement *Proc. 15th Int. Symp. on Fault-Tolerant Computing (Ann Arbor, MI)* (New York: IEEE) pp 200–6

[5] Kaashoek M F, Tanenbaum A, Hummel S F and Bal H 1989 An efficient reliable broadcast protocol *Operating Syst. Rev.* **23** (4) 5–19

[6] Molloy M K 1989 *Fundamentals of Performance Modelling.* (New York: Macmillan)