Originally published at pp. 608-12 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 2, 1994.

The Amoeba multicast protocol

This section describes the reliable multicast protocol developed by Kaashoek and Tanenbaum for group communication in Amoeba. This is a totally ordered multicast protocol that can be configured to provide a range of degrees of reliability [Kaashoek *et al*. 1989]. It employs a sequencer to order multicasts totally, and it uses the FLIP layer (see the description of Amoeba in the supplementary material at www.cdk3.net/oss), which employs hardware multicast to minimize the number of messages transmitted when multicasting over a local area network. The main achievement of this protocol is that in the normal case where no messages are lost, a multicast requires only two messages. We first describe the simplest version of the protocol, and then show how it can be extended to recover from computer failure.

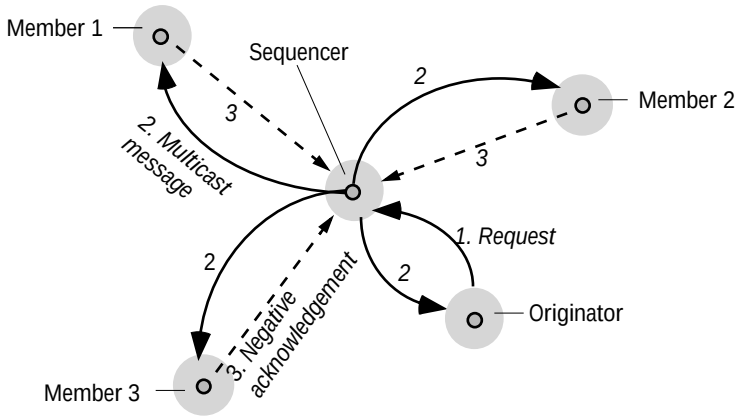
If all multicast messages are funnelled through a single member of a process group, that member can assign message identifiers from a sequence. The funnelling process is called the *sequencer*. Figure 1 illustrates the role of the sequencer in the transmission of a multicast message. The originator of a multicast message sends it to the sequencer which adds a sequence number and then relays it to the other members using a single broadcast message. The sequence numbers are used to ensure that multicast messages are delivered in the same order to all members.

The components that take part in the protocol are the members of a process group and the communication kernels that reside at each computer hosting a group member. The kernels implement multicast. At any one time, just one of the kernels acts as the sequencer. Several members of a group may run on the same computer. It is a straightforward matter to distribute a multicast message arriving at a computer to all the local group members. We therefore concentrate on multicasting a message to a set of destinations, where by 'destination' is meant computer.

In the simplest version of the protocol, the sequencer keeps the following information:

A *list of destinations* for the multicast messages (we assume that there is only one group and therefore one such set of destinations, but there could be more);

Figure 1 Multicast with a sequencer.



Note: the dashed arrows indicate (infrequent) requests for missing messages.

A *sequence number* that is incremented by one for every new multicast message. The sequence numbers are used to ensure that request messages are delivered in the same order to all the multicast destinations;

A *history buffer*, which holds a list of messages already sent to the destinations, together with their sequence numbers.

The originator of a multicast attaches a unique identifier to the message and transmits it point-to-point to the sequencer. The sequencer increments the sequence number and appends it to the message, which it stores in the history buffer. Then it transmits the message to all of the destinations. Hardware broadcast or multicast is used by the sequencer, assuming it is available, allowing the same network packet to be used to reach the multicast destinations and to acknowledge the originator's message. Otherwise, point-to-point messages would have to be used.

The originator times out and retransmits the message as necessary until it receives an acknowledgement. The sequencer checks for repeated messages against its history buffer, and merely sends an acknowledgement to the originator if the message is found there.

By contrast, the communication between the sequencer and the destinations uses a negative acknowledgment scheme in which the destinations request retransmission of lost messages from the sequencer. Lost messages are detected when a message arrives with a higher sequence number than is expected by the recipient. This is an example of a protocol designed to perform best under the normal conditions for local area networks – that is the loss of messages is infrequent. On the other hand, if a message does not arrive at a member that has stopped multicasting, then it won't find out that the message is missing until it sends a 'heartbeat' message to the sequencer, as discussed below.

Unless precautions are taken, the number of messages stored at the sequencer will grow indefinitely. The protocol takes the following steps to ensure that the history buffer capacity is not exceeded:

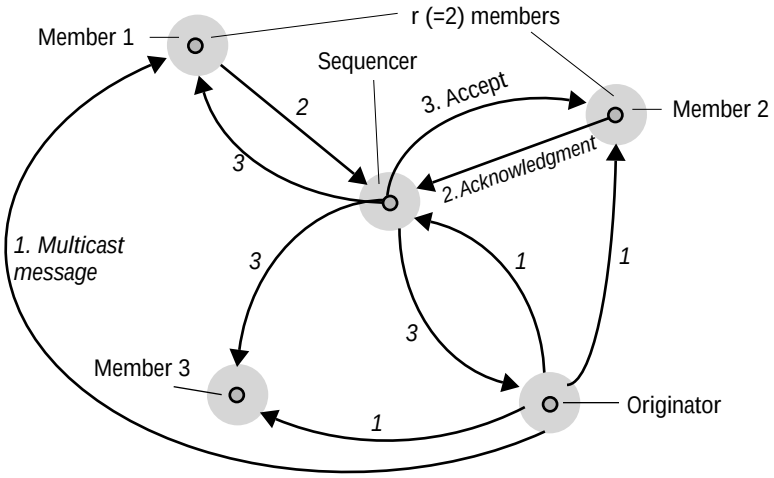
1. The highest sequence number received by the originator is piggy-backed on all the multicast messages. This enables the sequencer to record the highest message identifier seen by each member. Messages are removed from the history buffer after they have been acknowledged by all members of the group.
2. To ensure that there are regular acknowledgments even when members are not originating multicasts, each member is expected to send periodic 'heartbeat' acknowledgments of the highest sequence number received.
3. If the space occupied in the history buffer exceeds a pre-defined limit, the sequencer enters a synchronization phase during which no multicasts are done and the sequencer requests and ensures that all the members fetch any outstanding messages. After that the sequencer deletes the history buffer's contents and resumes its normal role.

Consider the case of a single group member multicasting repeatedly, with no other communication by group members. This is a worst-case scenario from the point of view of deleting messages from the history buffer. If the history buffer can hold h messages and if there are N group members, then after h messages are sent each of $N - 1$ members will have sent a heartbeat message, as mentioned above, in order that the history buffer is not filled. Assuming hardware multicast or broadcast is used, the number of packets required for h messages is $2h + N - 1$, and the number of packets per message is therefore $2 + (N–1)/h$. This number is liable to be close to 2, for a reasonably-sized history buffer.

One potential problem with this protocol is that the sequencer may become a bottleneck when the number of group members becomes large and multicasting is frequent. The results of a theoretical analysis are quoted by Kaashoek and Tanenbaum [1991] who suggest that this would be a problem at around 400 nodes for the most multicast-intensive applications they tried, although they do not specify what this is.

The most serious deficiency of this protocol as described is that its behaviour is unsatisfactory for many applications under conditions of failure.

Figure 2 Fault-tolerant multicast with a sequencer.



Reliability measure ◊ Kaashoek and Tanenbaum [1991] describe two ways in which the protocol described above is extended to take account of computer failures. First, the application is offered a chance to reconstitute the group and continue execution after computer failure is detected, even if the sequencer has failed. Secondly, multicast messages are guaranteed to be delivered to all surviving group members, despite up to *r* simultaneous computer failures, where the value of *r* is chosen by the user. If *r* is less than the number of members in the group, then multicast is not atomic. But the user can trade off this 'degree' of atomicity against what turns out, not surprisingly, to be the increased expense of the protocol as *r* increases.

When a computer fails in a fail-stop mode, one or more kernels notice this by virtue of its failure to respond. From that point, all group-related primitives are caused to fail. It is left up to the application to notice this and to respond. Any member can reconstitute the group by calling a special primitive, *resetGroup*. The effect of this primitive is to reestablish and report the group membership, and to once more enable multicasts to take place – so long as the group population exceeds a minimum number specified by the user.

When a group member fails, it is necessary to agree a new group membership list, agree on the kernel which is to act as the sequencer and to agree a new *incarnation number*. The incarnation number is effectively a group view identifier. Messages bearing an old incarnation number will be rejected by the sequencer.

To recover from the failure of the sequencer, the history buffer is replicated at all member sites. Each site retains copies of the messages it receives, and only deletes them when informed by the sequencer that they have been received everywhere (this information can be piggy-backed on normal message traffic). When a computer failure occurs, all computers at which *resetGroup* has been called send an *invitation* message to all other members of the group. In response, they send the highest sequence number received. Repeated attempts are made to gain a response from each computer, if necessary. The coordinator with the highest sequence number is elected as the final coordinator (the one with, say, the highest computer identifier is chosen in the case of a draw). The coordinator then fetches any missing messages (remember that not all computers contend to be the coordinator) and sends a *result* message containing the new incarnation identifier, the group membership and the highest sequence number to all members.

On receipt of the *result* message, the other kernels collect any missing messages from the coordinator. Once this is done, they send an *acknowledgement* message to the coordinator and resume normal operation. The coordinator resumes normal operation and acts as the new sequencer after it has received all *acknowledgement* messages. If it is discovered after the result message is sent that another process has failed, the protocol starts again from the beginning.

Finally, there is the problem of atomicity. Under the protocol so far described, a message could be delivered to some proper subset of the group, all of which then fail. The remaining members will never receive the message. There is a version of the protocol that guarantees atomic

delivery despite up to a specified number *r* of computer failures. In this version of the protocol (shown in Figure 2), the exchange of messages is different.

The originator multicasts the message. The recipients regard the message as unstable (that is, as not yet eligible for delivery) and keep it on a hold-back queue until they receive an *accept* message from the sequencer containing the sequence number. Control is not returned to the message's sender until the *accept* message is received. On receiving a message, a member records it and then sends an acknowledgment to the sequencer. After the sequencer has received the message, it waits for *r* acknowledgments and then broadcasts an *accept* message containing the next sequence number. At that point, *r*+1 kernels hold the message; any *r* of them can fail, and the message will still be delivered to the remaining kernels. The price paid for this guarantee is a two-phase protocol, which introduces considerable extra latency.

Summary

The Amoeba multicast protocol employs a sequencer, which labels multicast messages with sequence numbers in order to totally order the messages. Where hardware multicast is used, the basic protocol involves little more than two messages for each multicast. The protocol relies on a negative acknowledgement scheme, so that a process that has missed a message may not discover this until a subsequent 'heartbeat' message is sent to the sequencer. This delay may not be acceptable for some applications. The implementors report that the sequencer does not appear to be a bottleneck for the applications that they tried. However, it is not clear whether a sequencer-based protocol can be used to implement an efficient multicast service that could be used simultaneously by many applications.

A version of the protocol exists that allows the programmer to select the degree of reliability of the multicast: that is, the number of computers that may fail before delivery to the remaining computers cannot be guaranteed. This protocol relies on replication of the history buffer, which stores a record of all messages received by the sequencer.