RTCAST: Lightweight Multicast for Real-Time Process Groups

Tarek Abdelzaher, Anees Shaikh, Farnam Jahanian, and Kang Shin

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*zaher, ashaikh, farnam, kgshin*}@*eecs.umich.edu*

ABSTRACT

We propose a lightweight fault-tolerant multicast and membership service for real-time process groups which may exchange periodic and aperiodic messages. The service supports bounded-time message transport, atomicity, and order for multicasts within a group of communicating processes in the presence of processor crashes and communication failures. It guarantees agreement on membership among the communicating processors, and ensures that membership changes (e.g., resulting from processor joins or departures) are atomic and ordered with respect to multicast messages.

We separate the mechanisms for maintaining consistency in a distributed system from those of maintaining real-time guarantees. The multicast and membership service may be used alone to support service that is not time-critical. An additional, separate real-time admission control layer provides on-line schedulability analysis of application traffic to guarantee hard real-time deadlines in a system with dynamically-changing loads. Moreover, we separate the concerns of processor fault-tolerance and link fault-tolerance by providing a protocol for supporting the former, regardless of the actual network topology and technology used to implement the latter. We provide the flexibility of an event-triggered approach with the fast message delivery time of time-triggered protocols, such as TTP [1], where messages are delivered to the application immediately upon reception. This is achieved without compromising agreement, order and atomicity properties. In addition to the design and details of the algorithm, we describe our implementation of the protocol using the x-Kernel protocol architecture running under RT Mach 3.0.

Key Words — real-time process groups, membership protocol, atomic multicast, real-time schedulability

The work reported in this paper was supported in part by the Advanced Research Projects Agency, monitored by the US Air Force Rome Laboratory under Grant F30602-95-1-0044. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the funding agency.

1 Introduction

Process groups are a widely-studied paradigm for designing dependable distributed systems in both asynchronous [2–5] and synchronous [1, 6, 7] environments. In this approach, a distributed system is structured as a group of cooperating processes which provide service to the application. A process group may be used, for example, to provide active replication of system state so that it is available even when a process fails. Process groups may also be used to rapidly disseminate information from an application to a collection of processes that subscribe to that service. Two key primitives for supporting process groups in a distributed environment are *fault-tolerant multicast communication* and *group membership*.

Coordination of a process group must address several subtle issues including delivering messages to cooperating processes in a reliable (and perhaps ordered) fashion, maintaining consistent views of group membership, and detecting and handling process or communication failures. In a fault-tolerant system it is critical that members of the group maintain a consistent view of the system state as well as the group membership. Otherwise, members' actions in response to external or application-triggered events may be inconsistent, possibly resulting in serious consequences. Achieving consistency is therefore a paramount concern. Most process or processor group communication services achieve agreement by disseminating application and membership information via multicast messages within the group. If multicast messages are atomic and globally ordered, replicas can be kept consistent if we assume that process state is determined by initial state and the sequence of received messages.

The problem is further complicated in distributed real-time applications which operate under strict timing and dependability constraints. In particular, we are concerned here with fault-tolerant real-time systems which must perform multicast communication and group management activities in a timely fashion, even in the presence of faults. In these systems, multicast messages must be received and handled at each replica by their stated deadlines. In addition, membership agreement must be achieved in bounded time when processes in a group fail or rejoin, or when the network suffers an omission or a communication failure.

In this paper, we propose an integrated multicast and membership protocol that is suitable for the needs of hard real-time systems, and is also usable in a soft real-time system with synchronized clocks. It is termed *lightweight* because, in contrast to other group membership protocols, it does not use acknowledgments for every message and message delivery is immediate without the need for more than one "round" of message transmissions. We envision the multicast and group membership proposed here as part of a larger suite of middleware group communication services that form a composable architecture for the development of embedded real-time applications. Figure 1 shows a general framework within which the proposed protocol might be used. Shaded blocks indicate those services whose design and implementation we present in this paper. These services consist of two major components, a timed atomic multicast, and a group membership service. They are tightly coupled and thus considered a single service, referred to as *RTCast* in the remainder of the paper. Clock synchronization is assumed in the protocol and enforced by the clock synchronization service. To support portability, *RTCast* might lie atop a layer exporting an abstraction termed a *virtual network interface*. Ideally, this interface

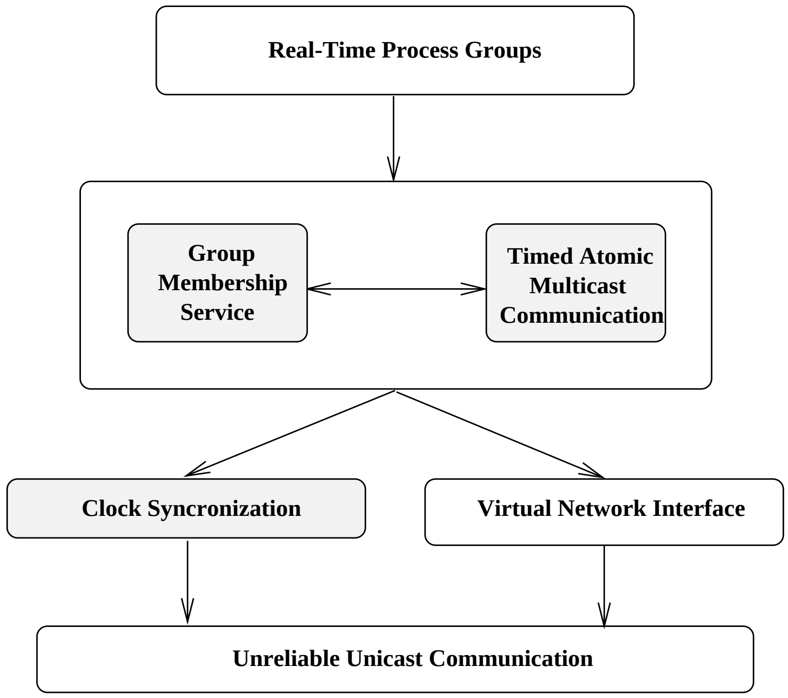

Figure 1: The general service framework.

would provide a mechanism to transparently handle different network topologies each having different connectivity and timing characteristics. In particular it quantifies the underlying network in terms of available bandwidth and maximum path delay between source/destination pairs, hiding its topology and technology. The network is assumed to support unreliable unicast. Finally, the top layer provides functional (API) support for the real-time process group service and interfaces to the lower *RTCast* protocol.

In its most basic form, *RTCast* proceeds as senders in a logical ring take turns in multicasting messages over the network. A processor's turn comes when the logical token arrives, or when it times out waiting for it. After its last message, each sender multicasts a heartbeat that is used for crash detection. The heartbeat received from an immediate predecessor also serves as the logical token. Destinations detect missed messages using sequence numbers and when a receive omission is detected on some processor, it crashes. Each processor, when its turn comes, checks for missing heartbeats, and eliminates the crashed members, if any, from group membership by multicasting a membership change message.

A key attribute of the way group membership is handled by *RTCast* is that processes which detect receive omissions take themselves out of the group. This paper argues that in a real-time system bounded message transmission time must be achieved. If a message (or its retransmission) doesn't make it to some destination within a specified jtime bound, the destination fails to meet its deadline(s) and should be eliminated from the group.

In the next section we discuss the related work on fault-tolerant multicast and group membership protocols. Sections 3 and 4 present our system model and the design of the *RTCast* protocol, respectively. The mechanisms used for real-time schedulability and admission control are described in Section 5. Section 6 discusses the current implementation of the protocol in the x-Kernel protocol development environment and our PC-based development testbed. Section 7 concludes the paper by discussing the limitations of this work and future research directions. In addition, we provide proofs of several protocol properties such as order, agreement, and atomicity in Section 8.

2 Related work

Several fault-tolerant, atomic ordered multicast and membership protocols have been proposed for use in asynchronous distributed systems. One of the earliest of these was the work done by Chang and Maxemchuk [8]. They proposed a token based algorithm for a group of communicating processes where each sender sends its messages to a *token site* which orders the received messages and broadcasts acknowledgments. Destinations receive the acknowledgments and deliver the corresponding messages in the order specified by the current token site. The token site is also responsible for handling retransmission requests originating from those destination nodes that detected receive omissions. To maintain proper order, a process does not send a message until it has received an acknowledgment of its previous one from the current token site. Though the algorithm provides good performance at low loads, the need for acknowledging each message before sending the next increases latency at higher loads. Moreover, introducing a third node (the token site) in the path of every message makes the service less available. The failure of the current token site will delay a destination's message reception even if both the source and destination are operational. By contrast, *RTCast* does not acknowledge each message, and need not involve an intermediate node on the path of each message.

ISIS [2, 9] introduced the concept of virtual synchrony, and integrated a membership protocol into the multicast communication subsystem, whereby membership changes take place in response to communication failure. ISIS implements a causal multicast service, CBCAST using vector clocks [10]. On top of CBCAST, an ordered atomic multicast, ABCAST, is implemented using an idea similar to that of Chang and Maxemchuk. A token holder orders received ABCAST messages and has the other members of the group deliver them in the same order. While we use the idea of integrating membership and multicast services, we implement the ordered atomic multicast directly without constructing a partial order first. The ordering task, however, is significantly simplified by assuming a token ring network. In addition to ISIS, several other systems have adopted the notion of fault-tolerant process groups, using similar abstractions to support distributed applications. Some of these include Consul [5], Transis [3], and Horus [4].

A number of systems choose to separate the group membership service from the fault-tolerant multicast service. As a result, the group membership service is concerned only with support for maintaining consistency regarding the membership view. These protocols may assume the availability of a separate reliable atomic multicast to support their operations. Our approach, along with several others described later, however, closely integrates the fault-tolerant ordered multicast and the group membership protocols. Achieving consistent membership when the group changes, is therefore no different than achieving consistent state in each of the processes.

The Strong Group Membership protocol [11], for example, performs atomic ordered group membership changes using a two phase commit protocol whereby a leader first multicasts a *prepare to commit* message, and then multicasts a *commit* message to install the change after receiving an acknowledgment from each group member. The MGS protocol [12] for processor group membership performs atomic ordered membership updates on three phases. The first phase requests/acquires a lock (by the group leader) on the membership state table of each member, the next sends the new data and collects acknowledgments, and the third phase retransmits changes to processors from which acknowledgments are missing. Additional work on group membership protocols appears in [13], [14], and [15].

Common to the above mentioned protocols, whether strictly group membership or combining multicast and group membership, is that they do not explicitly consider the needs of hard real-time applications. Thus these techniques are not suitable for the applications in which we are interested, namely those which require delay guarantees on group communication and membership management.

There are, however, several protocols that integrate reliable multicast and group membership and also target real-time applications. Some of these protocols are briefly described below. Totem [6] bears closer resemblance to the proposed protocol. It is based on a token ring, and guarantees *safe* delivery of messages within two token rounds (in the absence of message loss). Safe delivery means that a process does not deliver a message until it knows that every other process in the group has received it and will deliver the message unless it crashes. This ensures atomicity. The need for the second round is to convey acknowledgments that all processes have received the message sent during the first round. Totem addresses timing constraints by providing probabilistic bounds on message delay through analysis based on message loss probability. When message loss probability is very low, safe delivery within a predictable time is probable [16].

RTCast, on the other hand, achieves atomicity and order within a single phase. Messages are delivered to the application as soon as they are received in order. They may also convey membership changes for use by the protocol. No acknowledgments are needed. Furthermore, if one of the processors fails to receive a message and a retransmission request is issued, message delivery will be delayed only on that processor. The remaining ones can deliver immediately upon reception. This is in contrast with protocols like Totem where no processor can deliver a message until all have received it. The intuitive reason why immediate delivery does not interfere with atomicity in *RTCast* is that processors which have failed to receive a message (within the maximum amount of allowable retransmissions) will take themselves out of the group (i.e., crash) thus satisfying the definition of atomicity by default. A more formal argument is given in the Appendix.

Rajkumar *et. al.* [17] present an elegant publisher/subscriber model for distributed real-time systems. It provides a simple user interface for publishing messages on some logical "channel" (identified by a corresponding logical handle), as well as for subscription to selected channels as needed by each application. In the absence of faults each message sent by a publisher on a channel should be received by all subscribers. The abstraction hides a portable, analyzable, scalable and efficient mechanism for group communication. However, it does not attempt to guarantee atomicity and order in the presence of failures, which may compromise consistency.

TTP [1] is similar to the protocol presented in this paper in many respects. It uses a time-triggered scheme to provide predictable immediate message delivery, membership service, and redundancy management in fault-tolerant real-time systems. Unlike TTP, however, we follow an event triggered approach, where the complete event schedule need not be known *a priori*, and individual events may or may not be triggered by the progression of time. This, for example, makes it easier to accommodate prospective system growth beyond the initial design, without consuming off-line time for reinstallation. Moreover, while the design of TTP is simplified by assuming that messages sent are either received by all correct destinations or no destination at all (which is reasonable for the redundant bus LAN used in TTP), we also consider the case where a sent message is received by a proper subset of correct destinations. This might occur in the case of receiver buffer overflow, or message corruption on one of many links in an arbitrary topology network.

Finally, a research effort complementary to our efforts is reported in [18]. While we consider fault tolerance with respect to processor failure, we do not suggest any particular mechanism for implementing fault-tolerant message communication. For example, we do not specify whether or not some form of redundancy is used to tolerate link failures. On the other hand, Chen *et. al.* describe a combination of off-line and on-line analysis where spatial redundancy, temporal redundancy, or a combination of both, may be used to guarantee message deadlines of periodic message streams on a ring-based network in the presence of a number of link faults as specified for each stream. Unless the fault hypothesis is violated, the mechanism guarantees that at least one non-failed link will always be available for each message from its source to all destinations. Receive omissions might still occur because of data corruption and receiver buffer overflow, but these are handled by the multicast and membership algorithm proposed in this paper.

3 System model and assumptions

We consider a distributed system in which an ordered set of processing nodes $N = \{N_1, N_2, ..., N_n\}$ are organized into a logical ring. Each processing node $N_j \in N$ runs a set of processes which are members of a single multicast group. These processes communicate by multicasting messages to the entire group. The messages may or may not be periodic and may or may not have associated delivery deadlines. The order of processors on the ring determines the sequence in which they access the communication medium to send messages. The ring is assumed to have the following properties.

P1: Each processor N_j on the ring has a unique processor-id.

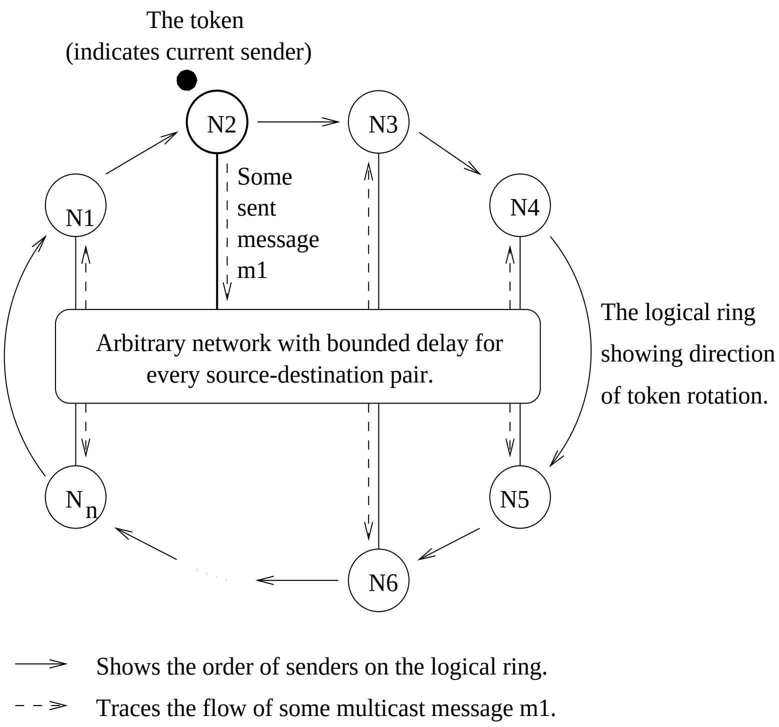

Figure 2: The logical ring.

P2: For any two processors, N_i, N_j, there exists a (logical) FIFO channel C_{ij} from N_i to N_j along which N_i can send messages to N_j (i.e., the network is not partitioned).

P3: Message delays along a channel are bounded by some known constant d_{max} unless a failure occurs. That is, any message sent along channel C_{ij} is either received within d_{max} or not received at all.

P4: Processor clocks are synchronized.

An illustration of the ring is shown in Figure 2. Note that total order of messages is trivially achieved because channels are FIFO, processors (senders) are ordered, and communication delay is bounded. All messages, unless they are lost, are received by each processor in the order they were sent. Thus, in the remainder of this paper we shall assume that if message m_i was received before message m_j then m_i was sent before m_j.

The assumed failure semantics are as follows.

A1: Processors fail by crashing, in which case the processor halts and its failure is detectable. A send omission is considered a special case of a Byzantine failure. It is converted to a crash failure by halting a processor if it does not receive its own message. Arbitrary or general Byzantine failures are not considered.

A2: Messages may be lost due to transient link failure, message corruption (in which case the corrupted message is discarded by the receiver), or receiver's buffer overflow, for example. In asynchronous systems we may also want to discard messages which are known to have violated property **P3**. Failures in which messages are discarded manifest themselves in the form of receive omissions. Permanent link failures (resulting in network partitions) are not considered. We believe that the proper way to handle permanent link failures in fault-tolerant real-time systems is to employ hardware redundancy, for example, as in TTP [1] or suggested in [18]. In this scenario at least operational link will remain and prevent a partition in the case of link failure. Thus, we do not consider partitions beyond the special case where one processor gets isolated from the network because of internal failure.

4 Multicast and membership service

Our primary purpose is to provide a multicast service for distributed real-time systems that achieves atomicity and total ordering on messages. Since order is trivially achieved under assumptions **A1** – **A3**, our main concern is to ensure multicast atomicity, so that "correct" members can reach consensus on replicated state. However, since we allow *both* message loss, and processor crashes, solving the consensus problem is impossible even in the presence of synchronized clocks [19, 20]. Instead, we formulate the problem as one of group membership. We employ a (deterministic) processor crash detection algorithm which occasionally eliminates a correct processor from the group so that agreement is reached in finite time among those remaining. This algorithm does not conflict with the consensus impossibility result, nor solve the consensus problem since the group is no longer the set of *all* correct processors. A processor that detects a receive omission takes itself out of the group. In a real-time system one may argue that processes waiting for the missing message on that processor will miss their deadlines anyway, so it is acceptable to eliminate that processor.

We may, however, provide at a lower communication layer support for a bounded number of retransmissions. In this case a message is declared as missing to the multicast protocol (i.e., a receive omission is detected) only after the maximum number of retransmission attempts is expired.

Exclusion of a processor on a receive omission is an easy way to guarantee atomicity. Each processor in the group either receives each multicast, or crashes. It is no longer required for a processor to know that every other processor has received the message before it can deliver it to the application (for example, as done in Totem's *safe* delivery [6]). Instead, each receiver may deliver the message *immediately upon receipt*, and be guaranteed that processors which do not receive that message will leave the group, thus preserving atomicty among (remaining) group members. Membership changes are communicated exclusively by *membership change messages* using our multicast mechanism. Since message multicast is atomic and ordered, so are the membership changes. This guarantees agreement on membership view. Order, atomicity and agreement are proved more formally in the Appendix.

Section 4.1 presents the steady state operation of the algorithm (with no receive omissions, processor crashes or membership changes). Section 4.2 then describes how receive omissions are detected and handled. Section 4.3 describes processor crashes and member elimination. Sections 4.4 and 4.5 discuss other membership changes (joins and leaves), and the relevant issue of recomputing token rotation time. Finally, Section 4.6 extends the design to manage message retransmissions.

Before continuing with the functional description of the proposed protocol, it is helpful to present its high-level structure. The protocol is triggered by two different event types, namely message reception, and token reception (or timeout). It is structured as two event handlers, one for each event type. The functions of these two handlers are presented below :

The **message reception handler** is invoked in the event of message reception. It's job is to detect receive omissions as described in Section 4.2, deliver messages to the application, and service messages directed to the protocol itself, e.g., join requests and resource reclaiming as described in Sections 4.4 and 4.5, respectively. Figure 3 gives the pseudo code for this handler.

The **token handler** is invoked when the token is received (i.e., a heartbeat is received from the predecessor in the ring), or when the token timeout expires. It is responsible for detecting processor crashes as described in Section 4.3 and for sending messages out as described in Section 4.1. Figure 4 shows the pseudo code for this handler.

Figure 3 and Figure 4 illustrate only detection of processor crashes and receive omissions, leaving out the details of other functionality of the handlers.

4.1 Steady state operation

We employ a token ring algorithm to control access to the communication medium. As shown in Figure 4, upon receipt of the token, a processor multicasts its messages starting with a *membership change message*, if any membership changes were detected during the last round. As messages are sent they are assigned successive sequence numbers by the sender.[1] The last message sent during a particular token visit is marked *last* by setting a corresponding bit. When the last message has been transmitted the processor multicasts a *heartbeat* (which has no sequence number). The heartbeat from processor N_i serves as an indication that N_i was alive during the given token visit (and therefore all its sent messages should be received). When received by its successor N_{i+1}, the heartbeat also serves as the logical token, informing the successor that its turn has come.

Each processor N_i has a maximum token hold time T_i after which it must release the token (that is, multicast the heartbeat). This guarantees a bounded token rotation time, P_{token}, which is important for message admission control and schedulability analysis. P_{token}[2] is given by :

$$P_{token} \; = \; \sum_{i=1}^{n} T_i \; + \; (n-1)d_{max}, \tag{4.1}$$

[1] Rollover is not a problem. Message number 0 follows in order the message with the maximum assignable number. Since message communication time is bounded, "old" messages do not survive to be confused with newer ones having the same number.

[2] Expression 4.1 will be refined in Section 4.4 as new factors are considered

```
msg_reception_handler()     /* invoked on any msg reception */
  if (state = RUNNING)
      if (sender = last_sender)          /* more msgs from the last sender? */
          if (sequence correct)          /* any missed msgs? */
              deliver msg
          else state = CRASHED     /* missed a msg */
      else if (sender≠last_sender)       /* new sender? */
          if (sequence correct)          /* any missed msgs? */
              missing_msg = FALSE
              for each processor p between sender and last_sender  /* didn't hear from p */
                  if NOT (msg_type = membership AND msg says p is OFF
                      AND p's last msg was received)
                      missing_msg = TRUE
              if (missing_msg = FALSE)
                  if (msg_type = membership) update membership view
                  else deliver msg
                  if (I was eliminated) state = CRASHED
             else state = CRASHED          /* missed a msg from some sender p */
         else state = CRASHED    /* missed a msg from new sender */
     else if (msg_type = joining)     /* somebody wants to join */
         handle join request
 if (state = JOINING AND msg_type = join_ack AND join_ack valid)
     if (need more join_acks)
         wait for additional join_ack msgs
     else state = RUNNING
end
```

Figure 3: The message reception handler.

where n is the number of processors in the current group membership, and d_{max} is as defined in **P3** in Section 3. If a processor fails or the token is lost, the bound on token rotation time can also be used as a timeout. If the token holder has no more messages to send before its hold time expires, it can release the token early. This prevents network bandwidth from being needlessly wasted by processors holding the token while no messages are being transmitted. Each processor *must* send at least one message during each token visit. If it has no messages to send, a dummy message is transmitted. This simplifies the detection of receive omissions, since each processor knows it must receive from *every other processor* within a token round, unless a message was lost.

4.2 Message reception and receive omissions

Each processor maintains a *message sequence vector*, M, which holds the sequence number of the last message received from every group member (including itself)[3]. Let M_i be the number of the last message

[3]This is different from using vector clocks [10] where a vector is communicated in each message. In our scheme, the message carries only its sequence number and sender id.

```
token_handler()                    /* invoked at token reception */
    for each processor p in current membership view
        if (no heartbeat seen from all my predecessors up to and including p)
            remove p from group view
            membership_change = TRUE
    if (membership_change = TRUE) send out new group view
    if (state = RUNNING)
        send out all queued messages and mark the last
        send out heartbeat msg
    if (state = JOINING)
        send out join msg
end
```

Figure 4: The token handler.

received from processor N_i. The multicast protocol layer expects to receive multicast messages in total order. Thus, after receiving message number M_i, the receiver expects message number $M_i + 1$ from N_i or, if M_i was marked *last*, the receiver expects message number $M_{i+1} + 1$ from processor N_{i+1}. N_{i+1} is the successor of N_i in the current group membership view. If the next message received, say m_k, is not the expected message, a receive omission is detected, and the receiver should crash.

As an optimization, we prevent receivers crashing upon detectably "false" receive omissions. Instead of having it crash, we first check whether or not the presently received message m_k is a membership change message which eliminates the sender, say N_j, of the missed message from membership[4]. If so, m_k also contains the number of N_j's last message sent before it was eliminated. This number is attached by the sender of the membership change message according to its own message vector information. If this number matches M_j in the current receiver's message sequence vector then the receiver is assured that all messages sent by N_j before it was eliminated have been received, and hence the receiver remains in the group. Otherwise, the receiver concludes that it did suffer a receive omission and it crashes. Lines $1 - 18$ of Figure 3 show how detection of receive omissions was implemented.

4.3 Membership change due to processor crashes

Each processor N_i keeps track of all other group members from which it has received a heartbeat during the current token round. That is, it records the processors from which it received a heartbeat since the time it sent its own and until it either receives that of its predecessor or times out, whichever comes first. Either case indicates that N_i's turn to send messages has come.

When N_i's turn comes, it first determines the processors from which it has *not* received a heartbeat within the last token round. A possible decision then is to assume that all of them have crashed and

[4] We know who the sender is because we know from whom a message is missing

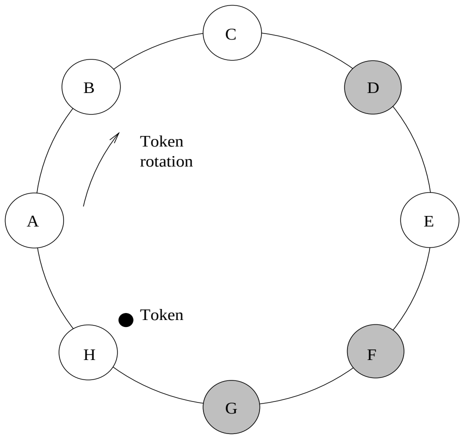

Figure 5: Excluding failed members.

eliminate them from group membership (by multicasting a corresponding membership change message). It turns out, a better decision is to eliminate only the transitive closure of immediate predecessors from which a heartbeat has not been received, if any. The rationale for this is best illustrated by an example. Consider Figure 5 where processor H has just timed out. Assume that H determines that it has not received a heartbeat from processors D, F, and G. In this case H should eliminate only F and G. D is not eliminated since it would have been eliminated by E had it indeed been down. If E has not eliminated D, then it must have received a heartbeat from it, and D must be alive, even though its heartbeat did not reach H. This idea is implemented as shown in lines $1-3$ of Figure 4.

As mentioned earlier, the detected membership change is effected by sending a membership change message. Crashes should be announced as early as possible. Therefore membership change messages precede all other messages sent by the processor during the token visit. Note that heartbeat loss might still lead on occasion to the exclusion of a correct processor from group membership since we generally cannot distinguish between that and a processor crash. Similarly, since successive heartbeats also serve as the rotating token, token loss and processor failure are treated in the same manner. As shown on line 16 of Figure 3, if a correct processor receives a membership message telling it that it has been excluded from the group it crashes.

4.4 Membership change due to processor join or departure

In the previous section we described how processors suspected of crashing are eliminated from the group. The remaining membership changes are voluntary member joins and member departures. A member can leave the group simply by multicasting a membership change message eliminating itself from membership. When a new processor, N_{new}, wants to join a group it starts out in a *joining* state where it sends a *join request* message to some processor N_p in the group, which may later multicast a membership change message on behalf of the joining processor, adding it to the group. The message is received atomically by members of the group who then send acknowledgments to the joining processor, containing their current membership view (with the new member added). The joining processor, now considered a member, checks that all received acknowledgments (membership views) are identical, and that acknowledgments have been received from every member in that view. If the check fails the new member crashes and attempts to rejoin later.

Note that, before joining, the new processor does not have an assigned slot on the ring. To address this problem the group contains a *join slot*, T_v, large enough for sending a join message. Expression 4.1, which gives the maximum token rotation time is thus refined to include T_v as follows :

$$P_{token} \;=\; \sum_{i=1}^{n} T_i \;+\; (n-1)d_{max} \;+\; T_v, \tag{4.2}$$

Our protocol is currently implemented on a broadcast LAN, so when the member following T_v on the ring receives the token from the member before T_v it waits for a duration T_v before sending its own messages. The slot T_v is used by a joining processor to send out its join request. If two processors send join requests at the same time, their requests will collide and will not be received by the destination. The joining processors will then wait for a random number of token rounds and retransmit their join. A joining processor need not know the position of T_v with respect to the current group members. Instead, it waits for a token with the *join slot bit* set on. The join request message, sent to processor N_p, contains the identity of the joining processor, and the requested maximum token hold time T_{new}.

Processor N_p who receives a join request computes the new P_{token} from 4.2, then initiates a *query round* in which it multicasts a query message asking whether or not the new P_{token} can be accommodated by each group member, and then waits for acknowledgments. If all acknowledgments are positive, N_p broadcasts the membership change adding the new member to the group.

4.5 Token rotation time

Each processor keeps a copy of variable P, which ideally contains the value of the maximum token rotation time P_{token} given by 4.2. P is used in admission control and schedulability analysis. When a processor fails and leaves the group, P_{token} decreases, since the number of processors on the ring is reduced. The failed processor will often try to rejoin very soon. Thus, to avoid updating P twice in a short time we may choose to keep the old value for a while after a processor crash.

P is thought of as a resource representing how much "space" we have available on the ring for processors to take. Each processor N_i is thought to take T_i out of P. Thus, P_{token} represents the present utilization of resource P. A joining processor may be added to the ring without the *query round* described in the previous section, if P_{token} (with the joining processor added) is still no greater than P. Otherwise, the query round is necessary to give all members a chance to check whether or not they can still guarantee their connections' deadlines under the new P. Schedulability analysis is described in Section 5.

P is updated by multicasting a corresponding message. In the case of a join, the membership message implicitly serves that purpose. In case of a member leaving or crashing, a separate message is used to update P. The point in time that the message is sent is a matter of policy. In the current implementation, a *resource reclaimer* module runs on the processor with the smallest id among those in the current membership. Note that since processors agree on current membership, they also agree on who runs the reclaimer. The module detects if any balance $P - P_{token}$ remained unutilized for more than a certain amount of time, after which it multicasts a request to reduce P by the corresponding balance.

4.6 Handling retransmissions

We mentioned earlier that a communication layer below the multicast and membership protocol, *RTCast* is responsible for message retransmission. Let us call it the *Retransmission* layer. A maximum number of allowable retransmissions r is specified for the system (to bound message delivery time). The retransmission layer checks for receive omissions the same way it was described in Section 4.2. If no receive omissions are detected, the *Retransmission* layer simply forwards the received messages up to the *RTCast* layer. Otherwise, it queues all messages subsequent to the detected gap in an incoming queue Q_I and queues a retransmission request in the outgoing queue. Also the receiver initializes a counter to the maximum number of allowable retransmissions, r. At each subsequent token visit the counter is decremented. If at any time a retransmitted message is received and the gap at the head of Q_I can be filled by the received message, the handler forwards all messages in Q_I up to the *RTCast* layer until the next gap, if any. If the counter is decremented to zero and no retransmission have been received to fill the gap at the head of Q_I, the next message is forwarded. The *RTCast* layer detects a receive omission, and the node crashes as was described in Section 5. As shown in Section 5, increasing the maximum number of allowable retransmissions means less messages can be guaranteed to meet their deadlines, since the worst case scenario must be accounted for in schedulability analysis. On the other hand the probability of a receive omission is significantly decreased. This is important in our algorithm, since receive omissions are translated into processor crashes.

5 Admission Control

In this section we discuss the admission control and schedulability analysis of real-time messages in the context of the multicast algorithm presented in the preceding section. In a real-time system this module implements a protocol layer above the *RTCast* layer to regulate traffic flow, although an application may choose to send messages directly to *RTCast* if hard real-time guarantees are not required.

Real-time messages may be either *periodic* or *aperiodic*. A periodic real-time message m_i is described by its maximum transmission time C_i, period P_i and deadline d_i, where $d_i \leq P_i$. An aperiodic message may be viewed as a periodic one whose period tends to infinity. Before a real-time message is considered for transmission its deadline must be guaranteed. Guaranteeing the deadline of a periodic message means ensuring that the deadline of each of its instances will be met, provided the sender and receiver(s) do not fail, and the network is not partitioned. The same applies to guaranteeing the deadline of an aperiodic message except that, by definition, the message has only one instance. Each message instance has an arrival time, which is the time at which the message is presented by the application. When an application needs to send a real-time message it presents the message to the admission control layer for schedulability analysis and deadline guarantee evaluation. The layer checks whether or not the message can be scheduled alongside the currently guaranteed messages, denoted by the set G, without causing itself or another message to miss its deadline. If so, the message is accepted for transmission and its deadline is guaranteed. Otherwise the message is rejected. Bandwidth is reserved for a guaranteed periodic message by adding it to set G, thereby affecting future guarantee decisions. The message is not removed from G until so instructed by the application. A guaranteed aperiodic message remains in set G only until it is sent. Non real-time messages are sent only after real-time messages, if time permits.

In order to perform schedulability analysis, we must make assumptions about the available network bandwidth. These are the following:

Assumption 1: Each sender node N_j can transmit messages for up to T_j units of time within any interval of time P.

Assumption 2: The interval elapsed between the time a message has been transmitted by the sender and the time it has been delivered at the destination(s) is bounded by some known constant $\triangle$.

The multicast algorithm presented in this paper satisfies both of **Assumption 1** and **Assumption 2**, with $\triangle = (r + 1)P$, where r is the maximum number of allowable retransmissions.

Note that **Assumption 1** does not state when and how network bandwidth becomes available for the node to send messages, nor does it assume any relation between message arrival times at the node and the time network bandwidth becomes available. Hence the analyses based on the above assumptions are general in nature and may be applied to a wide variety of multiple access network protocols.

Lemma 1: Under **Assumption 1** above, and FIFO scheduling of arrived messages, if the sum of transmission times, C_{total}, of messages arriving at node N_j within any period P is such that $C_{total} \leq T_j$, then each message will be sent within P units of time from its arrival.

Proof: We shall prove the complement form of the Lemma, that is, show that if a message is *late*, i.e., not sent within P units of time from its arrival at node N_j, then there exists some interval P during which $C_{total} > T_j$. Let message m_i, arriving at time t_1 at node N_j, be the first late message. Let t_2 be the time when the message has been transmitted. Thus, the interval $L_1 = [t_1,\ t_2]$ has length larger than P. By **Assumption 1**, the sum of transmission times of messages sent during L_1 is greater than T_j. Since these messages are sent FIFO, and m_i is sent last, all of them must have arrived prior to t_1. Furthermore, all of them must have arrived within an interval L_0 of length no more than P (prior to t_1), otherwise at least one of them would have been late too, which is impossible since m_i is the first

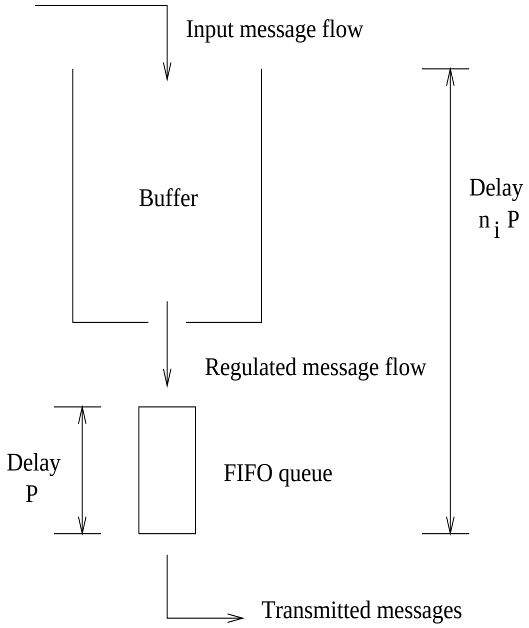

Figure 6: Message admission control.

late message. Thus there exists an interval of length P for which $C_{total} > T_j$ (namely the interval containing L_0).□

Lemma 1 derives a bound on message delay in the case where the sum of transmission times, C_{total}, of messages arriving at the sender node N_j within any period P is such that $C_{total} \leq T_j$. Next consider the case when the above condition is not satisfied. Let us call the sequence of messages presented by the application, the *input message flow*. The input message flow is passed through a buffer which produces a flow that satisfies the condition of Lemma 1, i.e., at most T_j worth of messages are produced within any period P. Let us call it the *regulated message flow*. Figure 6 is a conceptual view of the flow control mechanism. The following lemma determines a sufficient condition for a message to be schedulable given the described flow control scheme.

Lemma 2: Under **Assumptions 1 and 2**, a message m_i, which arrives at time t_0, can meet its deadline if :

- C1: $C_i \leq n_i\ T_j$, where $n_i = \lfloor (d_i - \triangle)/P \rfloor$,
- C2: The average input message flow F is no greater than T_j every P, i.e., $\sum_{i \in G} \frac{C_i}{P_i} \leq \frac{T_j}{P}$ and
- C3: at any time $t > t_0$, the sum of transmission times of all arrived messages in the interval $[t_0, t)$ is no less than $(t - t_0)F$ This merely ensures that the condition in Lemma 1 is always satisfied with no messages accumulating in the flow control buffer (provided that C2 is true).

Proof: Let t be the time message m_i arrived at node N_j. The quantity n_i represents the number of times the node received a transmission budget T_j between the arrival time of m_i and its deadline. Let the message m_i be divided into packets of length C_i/n_i. From condition C1 above, packet length is no more than T_j. Let these packets be made available for transmission at times t, $t + P$, ..., $t + (n_i-1)P$. From C2 and C3, the condition of Lemma 1 is met. Thus, by Lemma 1, each packet is transmitted within P units of time. Specifically, the last packet of m_i will be transmitted by $(n_i-1)P + P = n_iP$. Since n_i was chosen such that $n_iP \leq d_i - \triangle$, the message will meet its deadline.□

Next we present a sufficient condition for a collection of messages to meet their deadlines. This condition is used to check the schedulability of messages presented by the application and guarantee their deadlines. It is an extension of Lemma 2.

Theorem 1: A set of messages G presented by node N_j is schedulable if $\sum_{i \in G} \frac{C_i}{n_i} \leq T_j$

Proof: To prove the theorem it is enough to show that the conditions of Lemma 2 are satisfied for each message $m_i \in M$. C1 is trivially satisfied. To note that C2 is satisfied divide both sides of the inequality in the statement of the theorem by P, then note that the denominator of the left hand side $n_iP \leq d_i \leq P_i$. Thus, $\sum_{i \in G} \frac{C_i}{P_i} \leq \frac{T_j}{P}$, which satisfies C2. Finally note that in the worst case, all messages arrive together at some time t_0. Hence, it is easy to see that for any time $t > t_0$, the accumulated sum of transmission times of messages arrived within $[t_0, t)$ is $\sum_{i \in G} \lceil \frac{t-t_0}{P_i} \rceil C_i \geq (t - t_0) \sum_{i \in G} (\frac{C_i}{P_i}) = (t - t_0)F$. Thus condition C3 is also satisfied.□

To guarantee a new message or a periodic message stream, then, n_i is first computed for the message (stream). Then the sum indicated in Theorem 1 is calculated. If the inequality is satisfied the message (stream) is guaranteed. Otherwise, it is not guaranteed. Guaranteed messages or arriving messages belonging to guaranteed streams are forwarded to the *RTCast* layer for transmission over the network. Messages that cannot be guaranteed are rejected. Note that the set G always includes a periodic stream for sending membership messages with period P, and $n_i = 1$.

6 Implementation

The protocol presented in Section 4 is implemented on a network of Intel Pentium®-based PCs connected over a *private* Ethernet. Ethernet is not suitable for real-time applications due to packet collisions and subsequent packet retransmissions that may postpone successful packet transmission, and make it impossible to impose deterministic bounds on communication delay. However, since we use a *private* Ethernet (no one else has access to the communication medium), and since our token based protocol ensures that only one machine can send messages at any given time (namely, the token holder), *no collisions* are possible. The Ethernet driver will always succeed in transmitting each packet on the first trial. This makes message communication delays deterministic[5]. Note that some messages might be dropped at some destinations, for example, because of receiver buffer overflow, or data corruption.

[5]It is true however, that a collision may occur if two joining processes attempt to use the join slot simultaneously. For the sake of building a working prototype on the available hardware we presently circumvent this problem by preventing simultaneous joins

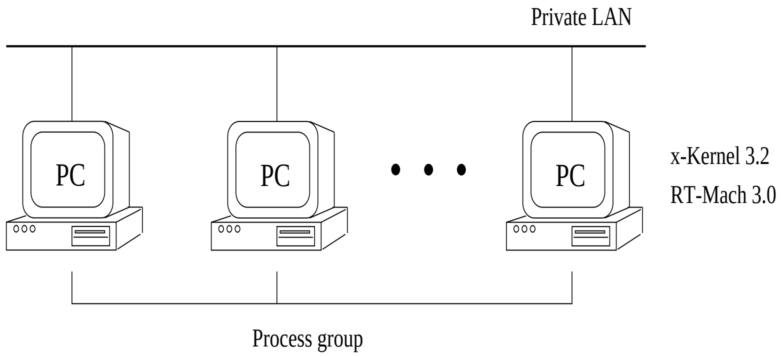

Figure 7: The testbed

This, of course, is not detected by the Ethernet driver. Instead, at a higher level, our protocol detects a message omission and may try a bounded number of retransmissions, r, as described in Section 4.6. These are accounted for as described in Section 5. In the present implementation, retransmissions are not supported. Each machine on the private LAN, runs the CMU Real-Time Mach 3.0 operating system. All machines are members of a single logical ring. Figure 7 illustrates the testbed.

6.1 Protocol stack layers

We implement the communication service as a protocol developed with x-Kernel 3.2 protocol implementation environment. The protocol stack is shown in Figure 8. Each box represents a separate protocol layer. The primary advantage of using x-Kernel is the ability to easily reconfigure the protocol stack according to application needs by adding or removing corresponding protocols.

The maximum functionality is attained by configuring the protocol stack as shown in Figure 8−(a). The *ACSA* layer performs admission control and schedulability analysis (as described in Section 5) to guarantee hard real-time deadlines of dynamically arriving messages and periodic connection requests. The *RTCast* layer implements the multicast and membership service as described in Section 4. The *Retransmission* layer is responsible for handling retransmissions as described in Section 4.6. ClockSync implements a clock synchronization service. In our present system, *ClockSync* uses the probabilistic clock synchronization algorithm developed by Cristian [21]. It uses the underlying unreliable messaging service provided in the x-Kernel environment.

In soft real-time systems, non real-time systems, or systems where hard real-time communication has been prescheduled and guaranteed *a priori*, we may wish to omit the *ACSA* layer, in which case the application interfaces directly to *RTCast*. The *RTCast* layer provides a subset of *ACSA*s API including message `send` and `receive` calls, but does not compute deadline guarantees.

If the underlying network is sufficiently reliable, we may choose to disallow message retransmissions deeming message omission very unlikely. This will enable supporting tighter deadlines, since we need not account for retransmissions when computing worst case deadline guarantees. This can be done by

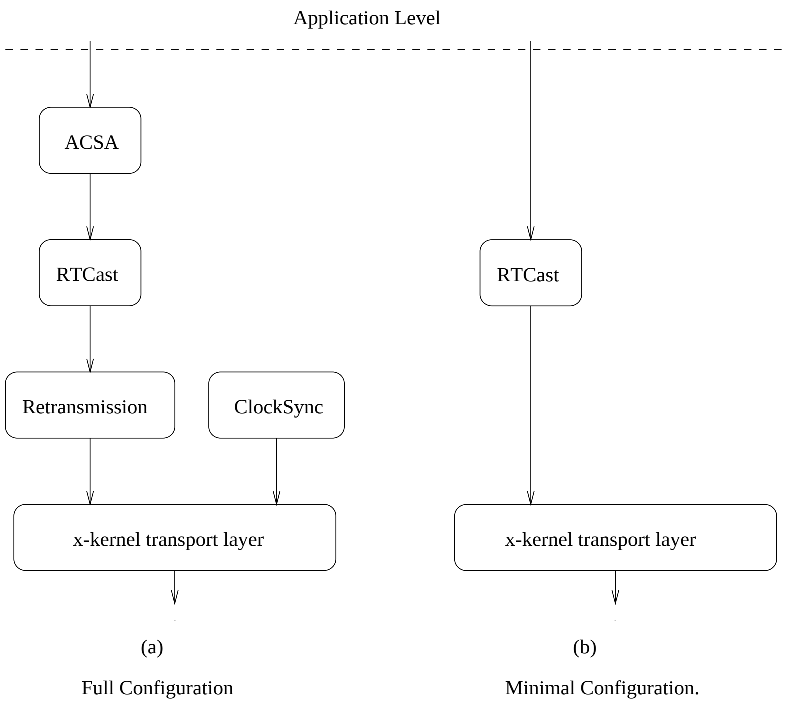

Figure 8: The x-kernel protocol stack

removing the *Retransmission* layer from the protocol stack[6].

Finally, note that *ClockSync* is needed only to implement assumptions **P2** and **P3** (Section 3). In the special case of a broadcast LAN, network hardware satisfies these assumptions obviating the need for *ClockSync*.

Thus, the minimal configuration of the protocol stack consists of the *RTCast* layer alone, as shown in Figure 8–(b). This configuration, when used on a broadcast LAN, supports bounded message transport delays, provides atomic ordered multicast, and implements a group membership service that guarantees atomic ordered membership changes, and agreement on group membership view.

6.2 *RTCast* message types

As mentioned in Section 1, *RTCast* is the essential layer of our group communication service. It is implemented by the two event handlers, **message reception handler** and **token handler**.

Figure 9 shows the different types of messages, and their formats, as well as the header format. Only the *atomic ordered* message types and the heartbeat/token type are illustrated. Atomic ordered types refer to messages guaranteed to be multicast atomically and in total order. When implementing the protocol we found it useful to support *unreliable* messages too, whose atomicity and order need not be guaranteed. These messages simply do not have sequence numbers. Thus, their omission is

[6] Presently, the *Retransmission* layer is not implemented

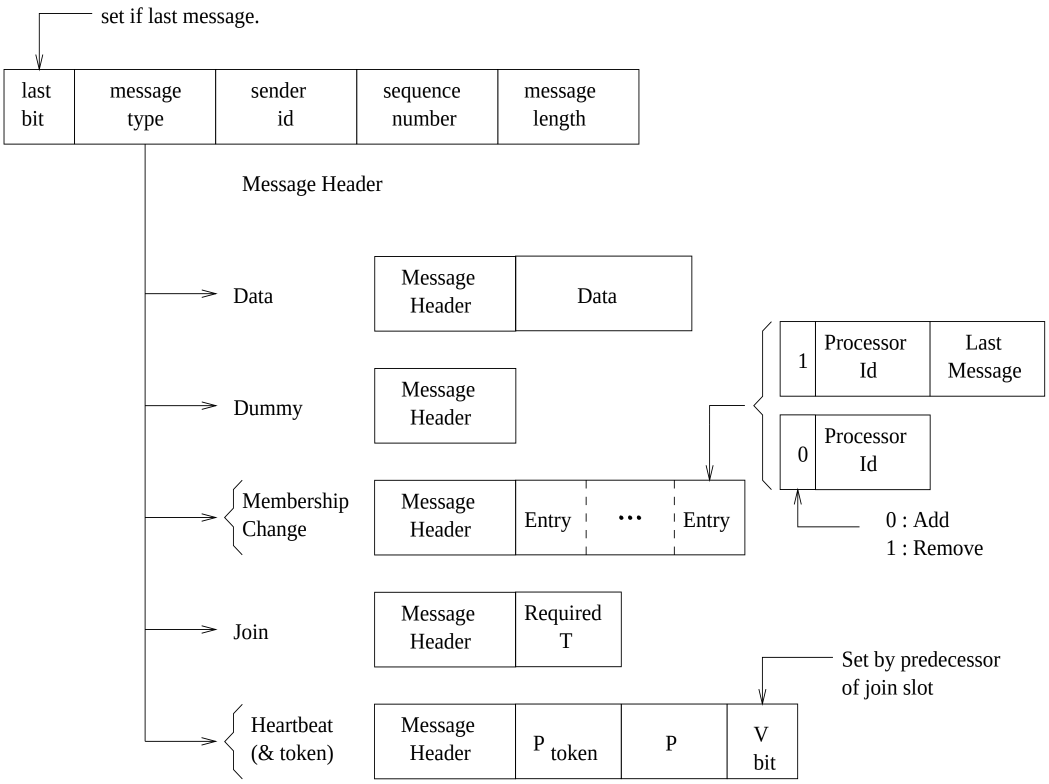

Figure 9: Message types

undetectable, and their order is not specified. For example, they are useful to implement voting. A given sender suggests a "proposition", then waits to receive "votes" from other group members before deciding whether or not some change should be (atomically) enacted. An instance of voting in our algorithm is to decide on a new member join when the current token rotation time P needs to be increased.

6.3 Protocol testing

Preliminary testing was performed to verify experimentally the behavior of the implemented protocol layers. The *RTCast* layer was tested first to verify its support for system consistency, then the *ACSA* layer was added, and the system was tested for deadline guarantees.

The *RTCast* protocol was tested employing the x-Kernel *trace library* to log the occurrence of certain major events at run-time. Major events include: sending and receiving of messages and heartbeats, token receipt (or timeout), "rotation" of current sender, detection of receive omissions and processor crashes, membership changes resulting from processor crashes, joins and leaves, and processor state transitions (between the *running*, *crashed* and *joining* states).

The system was run with event logging on the current testbed. Logs were then verified for conformity to intended semantics. Processor crash and receive omission failures where instrumented by introducing a uniformly distributed random variable for each type of instrumented failure. The current value of each variable was used to determine if the corresponding failure should be introduced next. These values were computed periodically. The probability of each failure was controlled by specifying the subrange of the corresponding random variable for which the failure is introduced. Crash failures were introduced by letting the failed processor go to the *crashed* state, from which it later recovers into the *joining* state to rejoin the group. Receive omission failures were introduced simply by dropping the next incoming message. Logs were then manually checked for order and atomicity of multicasts and membership changes, as well as agreement on membership view. In a subsequent set of experiments, the instrumented failures themselves were logged too, and logs were checked for correct failure detection as well.

To test the real-time behavior of the system, the protocol stack was configured with both *ACSA* and *RTCast* (with instrumented failures) present. Trace statements where used in the *ACSA* layer to record message arrival times and deadlines. The *RTCast* was used to log the receipt time of each message. It was verified that all messages guaranteed by *ACSA* made it to all destinations by their respective deadlines, unless the destination crashed. The messages themselves, in all experiments, were generated synthetically.

7 Conclusions

In this paper we presented *RTCast*, a new multicast and membership protocol to support fault-tolerant real-time applications. Our approach follows the process group paradigm in which a group of cooperating processes perform application tasks. We combine the flexibility of an event-triggered approach with bounded message transport and immediate message delivery upon receipt, without sacrificing order, atomicity, and agreement. In addition, *RTCast* supports on-line admission and schedulability analysis of periodic and dynamically arriving aperiodic messages. Finally, our implementation separates support for group management and fault-tolerant multicast from that of system timeliness by dividing functionality into two distinct layers, *RTCast* and *ACSA* respectively. The design is such that *RTCast* may be used alone if support for hard real-time guarantees is not required.

As with other group membership protocols we do not solve the distributed consensus problem; instead we occasionally conclude that an operational process has crashed. In this case the "crashed" process is forced to explicitly rejoin the group. Since our target domain is those applications which require hard real-time support, it is imperative that we provide failure detection and consistency in bounded time. As a result when messages are not received, or received late, we eliminate the receiver from membership. Similarly, when a token is not received by the specified time, we assume that the sending process has crashed. This allows remaining group members to reach agreement on membership and message order in bounded time. We argue that this is a valid tradeoff in a hard real-time system. Processes waiting for the omitted message would otherwise miss their deadlines, thus eliminating the processor is a natural thing to do.

As discussed in Section 1, *RTCast* represents part of a larger middleware service architecture providing group communication support for embedded real-time applications. As such, our current implementation realizes a subset of the suite of services outlined in Figure 1. We intend to improve the current preliminary implementation and continue toward the goal of developing a composable toolkit for provision of group communication support. Further areas of research on *RTCast* in particular include additional experiments to better characterize the performance and failure detection capability of the algorithm, investigation into techniques to exploit broadcast networks such as FDDI by fine-tuning the algorithms, and an extension of the protocol to support multiple process groups running on overlapping processors.

Acknowledgment

The authors wish to thank Jia-Jang Liou for his assistance in implementing the membership and clock synchronization services.

References

[1] H. Kopetz and G. Grünsteidl, "TTP – a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, pp. 14–23, January 1994.

[2] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, pp. 37–53, December 1993.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication sub-system for high availability," Tech. Rep. TR CS91-13, Dept. of Computer Science, Hebrew University, April 1992.

[4] R. van Renesse, T. Hickey, and K. Birman, "Design and performance of Horus: A lightweight group communications system," Tech. Rep. TR94-1442, Dept. of Computer Science, Cornell University, August 1994.

[5] S. Mishra, L. Peterson, and R. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs," *Distributed Systems Engineering Journal*, vol. 1, pp. 87–103, December 1993.

[6] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems*, vol. 13, pp. 311–342, November 1995.

[7] F. Cristian, B. Dancy, and J. Dehn, "Fault-tolerance in the advanced automation system," in *Proc. of Fault-Tolerant Computing Symposium*, pp. 6–17, June 1990.

[8] J.-M. Chang and N. Maxemchuk, "Reliable broadcast protocols," *ACM Transactions on Computer Systems*, vol. 2, pp. 251–273, August 1984.

[9] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems*, vol. 9, pp. 272–314, August 1991.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[11] F. Jahanian, S. Fakhouri, and R. Rajkumar, "Processor group membership protocols: Specification, design, and implementation," in *Proc. 12th Symposium on Reliable Distributed Systems*, pp. 2–11, 1993.

[12] L. Rodrigues, P. Veríssimo, and J. Rufino, "A low-level processor group membership protocol for LANs," in *Proc. Int. Conf. on Distributed Computer Systems*, pp. 541–550, 1993.

[13] Y. Amir, D. Dolev, and S. K. aned D. Malki, "Membership algorithms for multicast communication groups," in *Proc. 6th International Workshop on Distributed Algorithms*, no. 647 in Lecture Notes in Computer Science, (Haifa, Israel), pp. 292–312, November 1992.

[14] F. Cristian, "Reaching agreement on processor-group membership in synchronous distributed systems," *Distributed Computing*, vol. 4, pp. 175–187, 1991.

[15] A. M. Ricciardi and K. P. Birman, "Process membership in asynchronous environments," Tech. Rep. TR93-1328, Dept. of Computer Science, Cornell University, February 1993.

[16] L. Moser and P. Melliar-Smith, "Probabalistic bounds on message delivery for the totem single-ring protocol," in *Proc. IEEE Real-Time Systems Symposium*, pp. 238–248, 1994.

[17] R. Rajkumar, M. Gagliardi, and L. Sha, "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: Design and implementation," in *Proc. Real Time Technology and Applications Symposium*, (Chicago, IL), pp. 66–75, May 1995.

[18] B. Chen, S. Kamat, and W. Zhao, "Fault-tolerant real-time communication in fddi-based networks," in *Proc. 16th IEEE Real-Time Systems Symposium*, (Pisa, Italy), pp. 141–150, December 1995.

[19] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM*, vol. 34, pp. 77–97, January 1987.

[20] J. Turek and D. Shasha, "The many faces of consensus in distributed systems," *IEEE Computer*, vol. 25, pp. 8–17, June 1992.

[21] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, pp. 146–158, 1989.

8 Appendix

In this appendix the properties of the multicast algorithm, namely order, agreement and atomicity are proven. In the following *receiving* a message will signify the event of receiving it by the *RTCast* protocol layer. *Delivering* a message will refer to delivering it to the application. Let the membership view of some processor A be denoted V_A.

Lemma 3: The multicast algorithm guarantees order. That is, for any two messages m_i and m_j, if some processor A delivers m_i before it delivers m_j, then no other processor B, with $V_B = V_A$ delivers m_j until it has delivered m_i.

Proof: The property is proved by noting that messages are received in the order they were sent. More precisely, if processor A delivers m_i before m_j, m_i must have been sent before m_j. Thus m_i is either received before m_j by processor B, or is lost. In the former case m_i is delivered before m_j. In the latter case processor B detects message loss and identifies the sender(s) of the lost message(s) upon the receipt of the next message following the lost one(s) as described in Section 4.2. Processor B detecting a missing message m_i will crash, unless the sender of the lost message is not in V_B. But since $V_B = V_A$ and processor A delivered m_i, it must be that m_i is in V_A and thus is in V_B too. Hence, processor B crashes, i.e., it does not deliver m_j unless it delivered m_i. $\Box$

Next we prove agreement. That is to say, we shall prove that processors in a group always agree on group membership. First, consider the following lemma.

Lemma 4: For any two processors, A and B, if $V_A = V_B$ after *both* processors receive some message m_i, then $V_A = V_B$ after *both* processors receive any message m_j, such that m_j is sent after m_i.

Proof: We shall prove this lemma by induction on the number of messages sent within the multicast group. By induction hypothesis, let $V_A = V_B$ after both processors receive some message m_{j-n}. Let m_j be the next message received by *both* processors.

- If $n = 1$, no messages have been missed by either processor. Thus $V_A = V_B$ after receiving m_j.
- If $n > 1$, then messages m_{j-n+1}, ..., m_{j-1} have not been received by both processors. We have to cases:
  - All of these messages have not been received by *any* of the two processors. In this case we still have $V_A = V_B$ (unless one or both of these processors crash).
  - There exists some message received by one processor but not by the other. Let m_{j-k} be the most recent such message. Without loss of generality, assume A received it but B did not. In this case when B receives m_j, a receive omission is detected. Let C be the sender of m_{j-k}.

Since A received m_{j-k}, C must have existed in $V_A = V_B$ after m_{j-n}[7]. Thus, B will crash upon the receive omission from C (trivially satisfying the lemma) unless the sender of m_{j-k}, say C, was eliminated from V_B by the next message it received, namely message m_j. But if m_j eliminates C from membership then either A or B must crash upon receiving m_j. This is because a message, e.g. m_j, eliminating a processor from membership, e.g. C, also indicates the last message sent by the eliminated processor. A processor receiving the membership message m_j will crash unless it has received all messages from the eliminated processor, C, up to the one indicated in m_j. Since A and B have received a different number of messages from C (A received m_{j-k}, but B didn't) at least one of them will crash upon receiving m_j, satisfying the lemma.

The above cases prove the lemma □

Next we prove the following stronger result.

Lemma 5: For any two processors A and B, if $B \in V_A$ and neither processor has crashed, then $V_A = V_B$ after the current round of the token.

Proof : When first started, processor A makes a singleton group containing only itself. If $B \in V_A$, then one of following must have happened since A was started :

- Processor A joined a group containing B. Note that in order to join the group, processor A sends a *join* request and waits for *join_ack*s, which also contain group membership. Processor A joins the group, with the indicated membership, only if it has received, within a single round of the token, *join_ack*s with the same membership view from every processor indicated in that view, and if that view is precisely the set of processors from which the *join_ack*s have been received in addition to itself. This guarantees that all processors agree on group membership when a new one is admitted to the group. Thus, for any processor $B \in V_A$, we have $V_B = V_A$ which satisfies the lemma.

- Processor A receives B's *join* message: In this case, if B eventually joins the group, it must have received A's *join_ack* thus satisfying $V_A = V_B$, otherwise B does not become a member, and behaves as if it crashed. Thus, if $B \in V_A$, and neither processor crashed, then $V_A = V_B$.

The above two cases show that when B is added to A's membership view, both processors have the same view, $V_A = V_B$. By the Lemma 4, if $V_A = V_B$ is initially true, then $V_A = V_B$ remains true after the receipt of any message m_j by *both* processors unless one or both of the processors crash. Thus, to prove Lemma 5, it remains to note that if no messages are received by both processors for one round of the token, processor A will not have received a message from B during that round (B is assumed to

[7] If C joined the group after m_{j-n}, and since a new member does not join the group unless it has verified agreement on membership, then B knows of C too

receive its own messages). Therefore, either A crashes on a receive omission from B or it must be that $B \notin V_A$. In other words, if A hasn't crashed, and $B \in V_A$ then there exists a message received by both A and B in the last round in which case the lemma is satisfied. □

Corollary 1: For any two processors A and B, if $B \notin V_A$ and neither processor has crashed then $A \notin V_B$ after the current round of the token.

Proof: By contradiction, if $A \in V_B$ then, by previous lemma, after the current round of the token $B \in V_A$, but that is a contradiction since $B \notin V_A$. Thus $A \notin V_B$. □

The agreement property is a direct consequence of the above.

Lemma 6: The membership algorithm satisfies the agreement property. That is, every running processor belongs to some set G, such that for every running processor $B \in G$, $V_B = G$.

Proof: Consider some processor A with membership view V_A. Let G be the set V_A. By the previous lemma, for every processor B in V_A, since $B \in V_A$, then $V_B = V_A$. Substituting G for V_A, we prove the statement of the lemma. □

Next we prove atomicity.

Lemma 7: Every message m_i sent by some processor A to the processors in V_A, is delivered by every processor in V_A, unless it crashes.

Proof: Every processor B in V_A either receives the message, in which case it is delivered, or detects a message loss from A. In the latter case :

- If $A \in V_B$, then B will crash.
- If $A \notin V_B$, then by the corollary, $B \notin V_A$ after the current round of the token, i.e., from A's standpoint, B has crashed. This case happens when the network partitions. Processors on either side of the partition will think the ones on the other side have crashed.

This proves the lemma. □

Theorem 2: The proposed multicast and membership service guarantees order, atomicity, and agreement under the given assumptions and failure semantics.

The proof follows directly from Lemma 3, 7 and 6.