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 provide the flexibility of an event-triggered approach with the fast message delivery time of time-triggered protocols, such as TTP [14], 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 on RT Mach 3.0.

1. Introduction

Process groups are a widely-studied paradigm for designing dependable distributed systems in both asynchronous [6, 3, 24, 17] and synchronous [14, 4, 11] 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 or to rapidly disseminate information from an application to a collection of processes. 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 the group in a reliable (and perhaps ordered) fashion, maintaining consistent views of group membership, and detecting and handling process or communication failures. It is critical that group members maintain a consistent view of the system state and group membership to avoid inconsistent response to external or application-triggered events. 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 when 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 needing more than one "round" of message transmissions. We envision the proposed protocol as part of a larger suite of middleware group

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.

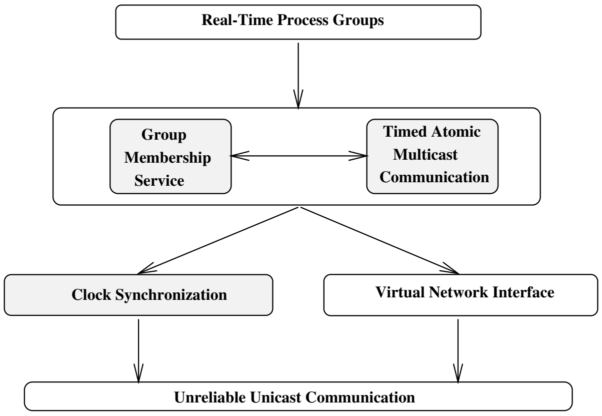

Figure 1. A general framework for a middleware group communication service

communication services that form a composable architecture for the development of embedded real-time applications. Shaded blocks in Figure 1 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 would transparently handle different network topologies, each having different connectivity and timing or bandwidth characteristics. 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.

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 processor detects a receive omission, 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 *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) does not reach a destination within a specified time 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. 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.

2. Related work

Several fault-tolerant, atomic ordered multicast and membership protocols have been proposed for use in asynchronous distributed systems. In some of the earliest work, Chang and Maxemchuk [7] proposed a token based algorithm for a process group where each member sends its messages to a token site which orders the messages and broadcasts acknowledgments. Destinations use the acknowledgments to order messages as specified by the token site. Though the algorithm provides good performance at low loads, 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. Failure of the token site will delay message reception even if both the source and destination are operational. In contrast, *RTCast* does not acknowledge each message, and need not involve an intermediate node on the path of each message.

ISIS [5, 6] 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 an atomic ordered multicast on top of a vector clock-based [15] causal multicast service, using an idea similar to that of Chang and Maxemchuk. We integrate membership and multicast services, but implement ordered atomic multicast directly without constructing a partial order first. The ordering task, however, is simplified by assuming a 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 [17], Transis [3], and Horus [24].

A number of systems choose to separate the group membership service from the fault-tolerant multicast service. As a result, the group membership service maintains consistency regarding the membership view and may assume that separate reliable atomic multicast support is available. Examples of this approach are found in the Strong Group Membership protocol [13] and the MGS protocol for processor group membership [21]. Additional work on group membership protocols appears in [2, 10, 20].

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. There are, however, several protocols that integrate reliable multicast and group membership and also target real-time applications.

Totem [4, 18] is an example of a protocol that provides probabilistic real-time guarantees. It is based on a token ring, and guarantees atomic ordered delivery of messages within two token rounds (in the absence of message loss). *RTCast*, on the other hand, achieves atomicity and order within a single round. Messages are delivered to the application as soon as they are received in order without the need for acknowledgments. 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. The intuitive reason why immediate delivery does not interfere with atomicity in *RTCast* is that processors failing to receive a message take themselves out of the group.

Rajkumar *et. al.* [19] present an elegant publisher/subscriber model for distributed real-time systems. It provides a simple user interface for publishing messages on a logical "channel", and for subscribing 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. It does not, however, attempt to guarantee atomicity and order in the presence of failures, which may compromise consistency.

TTP [14] is similar to *RTCast* 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 gain flexibility by following 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. Moreover, 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, which 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 ours is reported in [8]. While we consider fault tolerance with respect to processor failure, we do not suggest a mechanism for implementing fault-tolerant message communication. For example, we do not specify whether or not 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.

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 representing a single multicast group. Figure 2 depicts the ring configuration. Each node runs a daemon process responsible for multicast communication. The ring is assumed to have the following properties:

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

P2: For any processor pair (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.

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.

We achieve total order of messages and enforce timeliness using the above assumptions. The specific mechanisms are described in more detail in Sections 4 and 6. The failure semantics are as follows :

A1: Processors fail by crashing, in which case the processor halts and its failure is detectable. Send omissions are converted to crash failures by halting a processor if it does not receive its own message.

A2: Message receive omissions are allowed, e.g., due to transient link failures, or by discarding by the receiver, due to corruption or buffer overflow. 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 [14] or suggested in [8].

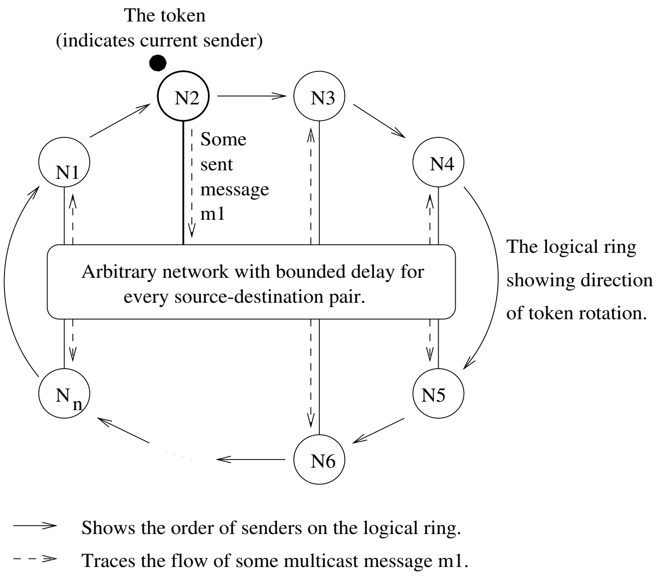

Figure 2. The logical ring

```
msg_reception_handler()
if state = RUNNING
 if more msgs from same member
  if missed msgs → CRASH else
   deliver msg
 else if msg from different member
  if missed msgs → CRASH else
   check for missed msgs from processors
    between current and last senders
   if no missing msgs
    deliver current msg
   else CRASH
else if join msg from non-member
 handle join request
if state = JOINING AND
 msg is a valid join_ack
 if need more join_acks
  wait for additional join_acks
 else state = RUNNING
end
```

Figure 3. Message reception handler

4. Multicast and membership service

Our primary purpose is to provide a fault-tolerant atomic ordered multicast service for distributed real-time systems that achieves agreement on replicated state. In a token ring, sent messages have a natural order defined by token rotation. Senders are logically organized in a ring and messages from each are sequenced in the order they are sent. We reconstruct message order at the receivers using a protocol layer below $RTCast$ which detects out-of-order arrival of messages and swaps them, thus forwarding them to $RTCast$ in correct order. Section 6.1 provides more details on this layer. Note that it does not guarantee reliability; lost messages will result in gaps in the forwarded sequence. $RTCast$ is designed to deal with such receive omissions. In the following discussion, message reception refers to reception by $RTCast$ unless otherwise stated.

$RTCast$ ensures atomicity so that "correct" members can reach agreement on replicated state. We formulate the problem as one of group membership. A processor that detects a message receive omission takes itself out of the group, thus maintaining agreement among the remaining ones. Each receiver may deliver each message *immediately upon receipt*, yet be guaranteed that delivery is atomic to all group members since processors which missed that message left the group. In a real-time system one may argue that processes waiting for a message that does not arrive will miss their deadlines anyway, so it is acceptable to eliminate the processor(s) which suffered receive omissions.[1] Our algorithm allows a processor to become inconsistent with the group but ensures that such inconsistencies are contained. The inconsistent processor will always crash before it communicates any messages. For the rest of the group this is identical to the case where no inconsistency ever arose, assuming no hidden channels.

[1] A lower communication layer may support a bounded number of retransmissions. See Section 4.6.

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 proven more formally in [1].

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.

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 **message reception handler** (Figure 3) detects receive omissions as described in Section 4.2, delivers messages to the application, and services protocol control messages. The **token handler** (Figure 4) is invoked when the token is received or when the token timeout expires. It detects processor crashes as described in Section 4.3 and sends messages out as described in Section 4.1.

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

Figure 4. Token handler

4.1. Steady state operation

We employ a logical token ring algorithm to control access to the communication medium. 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.[2] 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. A token holder releases the token (i.e., multicasts the heartbeat) when it has sent all its messages, or when T_i has expired, whichever comes first. This guarantees a bounded token rotation time, P_{token}, which is important for message admission control and schedulability analysis. It also makes it possible to set the timeout used to detect token loss. P_{token}[3] is given by:

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

where n is the number of processors in the current group membership, and d_{max} is as defined in **P3** in Section 3.

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)[4]. Let M_i be the number of the last message 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 crashes.

As an optimization, we prevent receivers crashing upon detectably "false" receive omissions. Instead of forcing a crash, we first check whether or not the just received message m_k is a membership change message which eliminates the sender of the missed message, say N_j, from membership[5]. 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.

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 eliminate them from group membership (by multicasting a corresponding membership change

[2]Rollover problems are avoided since message communication time is bounded; "old" messages do not survive to be confused with newer ones.

[3]Expression 1 will be refined in Section 4.4 as new factors are considered.

[4]This is different from using vector clocks [15]; here, messages carry only their sequence number and sender id.

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

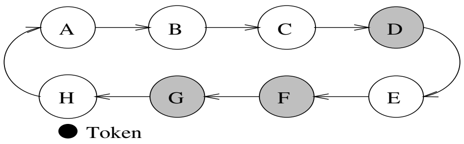

Figure 5. Excluding failed members.

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.

Note that this algorithm does not distinguish between token loss and processor failure. Thus, a correct processor may occasionally be eliminated from the group. If a processor receives a membership message telling it that it has been eliminated from the group it crashes.

4.4. Membership change due to joins and leaves

In the previous section we described how processors suspected of having crashed 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. On a multiple access LAN this causes a problem since any access to the communication medium has to be charged to a slot assigned to some processor in the group in order to preserve bounded token rotation time.[6] To address this problem in our broadcast LAN implementation, the group contains a *join slot*, T_v, large enough for sending a join message. Expression (1), which gave the maximum token rotation time is thus refined to include T_v as follows :

$$P_{token} \;=\; \sum_{i=1}^{n} T_i \;+\; (n-1)d_{max} \;+\; T_v, \qquad (2)$$

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. In the case of multiple simultaneous joins, processors that fail to send the join message successfully within the assigned common slot 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* set on (sent by the join slot's predecessor). 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 (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 (2). P is used in admission control and schedulability analysis. When a processor fails and leaves the group, P_{token} decreases, since there are now fewer processors on the ring. The failed processor will often try to rejoin very soon. Thus, to avoid updating P twice in a short time we may keep the old value for a while after a 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

[6] In a general topology network the computed value of d_{max} should account for the extra traffic.

serves that purpose. When a member leaves or crashes, a separate message is used to update P. The point at which 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. Since processors agree on current membership, they also agree on who runs the reclaimer. The module detects if any balance $P - P_{token}$ remained unused 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

A valid criticism of the presented algorithm is that it converts receive omissions into processor crash failures which allows rapid decomposition of the group when the probability of message loss is high. Permitting a bounded number of retransmissions, however, significantly reduces loss probability, eliminating this problem. As mentioned earlier, a communication layer below $RTCast$, the $Retransmission$ layer, may be responsible for (transparent) message retransmission. During normal operation this layer simply forwards all messages up to $RTCast$ in received order. However, if a message is detected missing it queues up an outgoing retransmission request and temporarily holds all subsequent incoming messages. If a retransmission is not received within a pre-specified number r of token rounds, the retransmission layer skips the missing message(s), and resumes forwarding the available ones up to $RTCast$ (which then detects a receive omission and reacts accordingly as described in Section 4.2).

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. 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 we assume that $d_i \leq P_i$. An aperiodic message may be viewed as a periodic one whose period tends to infinity. Before considering a real-time message 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 aperiodic message deadlines 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 G until it is sent. Non real-time messages are sent only after real-time messages, if time permits.

In order to perform schedulability analysis, and in view of the algorithm presented in the previous section, we can make the following assumptions:

Assumption 1: Each sender node N_j has a bounded token hold time T_j, and a bounded token interarrival time $P - T_j$.

Assumption 2: The elapsed interval 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 a known constant $\triangle$. (Note that $\triangle = (r + 1)P$, where r is the maximum number of allowable retransmissions.)

Under the above assumptions we have a sufficient schedulability condition, given by the following theorem. The proof of the theorem is detailed in [1].

Theorem 1 *A set of messages G presented by node N_j is schedulable if $\sum_{i \in G} \frac{C_i}{n_i} \leq T_j$, where $n_i = \lfloor (d_i - \triangle)/P \rfloor$.*

The above schedulability condition does not depend on details of the underlying communication algorithm as long as it provides bounds on token hold time, token interarrival time, and message communication delay. Similarly, $RTCast$ is unaware of the type of admission control policy used. The goal of this separation is to allow use of a number of admission control policies to ensure timeliness while leaving consistency to the multicast algorithm. For example, instead of the schedulability condition stated in **Theorem 1**, one may use the generalized rate monotonic analysis [22], FDDI synchronous bandwidth allocation analysis [16], or delay analysis for Controller Area Networks [23], depending on the application at hand.

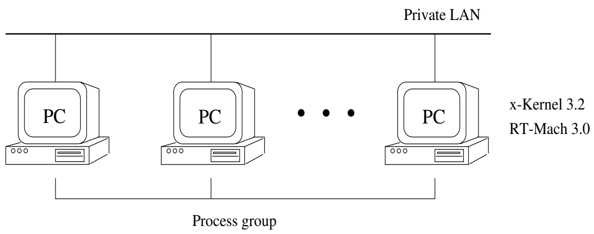

Figure 6. The protocol testbed

6. Implementation

The protocol presented in Section 4 is implemented on a network of Intel Pentium®-based PCs connected over a *private* Ethernet. In general, Ethernet is unsuitable for real-time applications due to packet collisions and subsequent retransmissions that make it impossible to impose deterministic bounds on communication delay. However, since we use a *private* Ethernet (exclusive 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 always succeeds in transmitting each packet on the first trial, making message communication delays deterministic[7]. Note that detection of dropped messages, due to buffer overflow or data corruption, is left to 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 and all machines are members of a single logical ring. Figure 6 illustrates the testbed.

6.1. Protocol stack layers

We implement the communication service as a protocol developed in the *x*-Kernel 3.2 protocol implementation environment [12]. The protocol stack is shown in Figure 7. 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.

Maximum functionality is attained by configuring the protocol stack as shown in Figure 7-(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 described in Section 4. The *Retransmission* layer is responsible for handling retransmissions as described in Section 4.6. $ClockSync$ provides a synchronization service using the probabilistic algorithm developed by Cristian [9]. It uses the underlying unreliable messaging service provided in the *x*-Kernel environment.

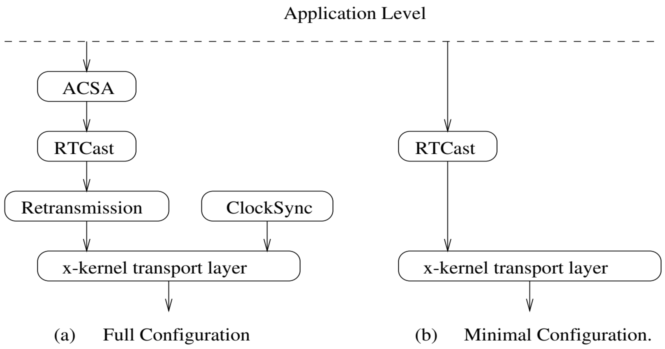

Figure 7. The x-kernel protocol stack

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 not to use message retransmissions. This will enable supporting tighter deadlines, since we need not account for retransmissions when computing worst case deadline guarantees. This can be done by removing the *Retransmission* layer from the protocol stack.

Finally, *ClockSync* is needed to implement the gap detection property and message order. In an arbitrary network, messages may be received out of order by the machine. A protocol layer, $Order$ may be used to reconstruct message order. We showed in Section 4.2 that upon message reception it is possible to detect whether it has arrived in correct order. For example, if a message with a timestamp t_{send} arrives before its preceding one, $Order$ may wait until $t_{send} + d_{max}$, before regarding the missing message lost and forwarding its successor(s) to $RTCast$. If the missing message arrives before $t_{send} + d_{max}$, it is forwarded immediately with the rest in correct order. Note that this mechanism requires the sender and receiver to have synchronized clocks. In the special case of broadcast LANs, however, messages are ordered by the communication medium and $Order$ is not needed. *ClockSync* is also needed for receivers to enforce timed multicast semantics, if so desired; that is to drop messages which arrive after their deadline. Soft real-time applications may not need this property.

Thus, the minimal configuration of the protocol stack

[7]A collision may occur, however, if two processes try to use the join slot simultaneously. We presently circumvent this problem by preventing simultaneous joins.

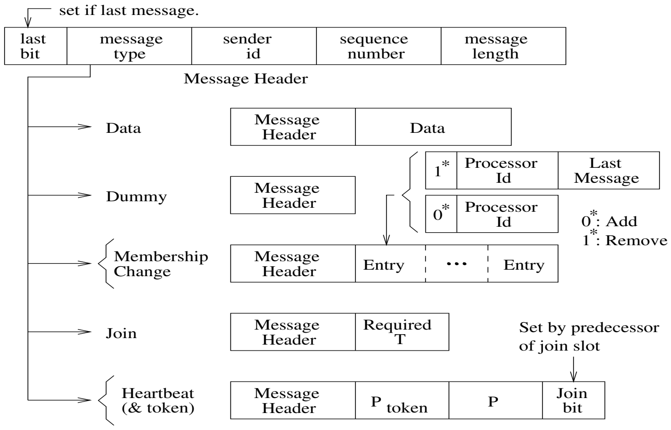

Figure 8. Message types

consists of the *RTCast* layer alone, as shown in Figure 7-(b). This configuration, when used on a broadcast LAN, supports bounded message transport delay, 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 8 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 are those messages guaranteed to be multicast atomically and in total order.

When implementing the protocol we also found it useful to support *unreliable* messages, whose atomicity and order need not be guaranteed. These messages do not have sequence numbers and thus, their omission is undetectable, and their order is not specified. They are useful to implement voting, for example. A given sender suggests a "proposition", then waits to receive "votes" from group members before deciding whether some change should be (atomically) enacted. An instance of voting in our algorithm is to decide on a new member join when the 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 using 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. More details regarding performance evaluation are reported in [1].

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 performs 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 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 and the reviewers for their valuable comments and suggestions.

References

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RT-CAST: Lightweight multicast for real-time process groups. Technical Report CSE-TR-291-96, Dept. of Elec. Engineering and Comp. Science, University of Michigan, January 1996.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proc. 6th International Workshop on Distributed Algorithms*, number 647 in Lecture Notes in Computer Science, pages 292–312, Haifa, Israel, November 1992.

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

[4] 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*, 13(4):311–342, November 1995.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[7] J.-M. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

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

[9] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

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

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

[12] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

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

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

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] N. Malcolm, S. Kamat, and W. Zhao. Real-time communication in FDDI networks. *Real-Time Systems*, 10(1):75–107, January 1996.

[17] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.

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

[19] 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*, pages 66–75, Chicago, IL, May 1995.

[20] A. M. Ricciardi and K. P. Birman. Process membership in asynchronous environments. Technical Report TR93-1328, Dept. of Computer Science, Cornell University, February 1993.

[21] L. Rodrigues, P. Verı́ssimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Proc. Int. Conf. on Distributed Computer Systems*, pages 541–550, 1993.

[22] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[23] K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, August 1995.

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