

# A Low-Cost Processor Group Membership Protocol for a Hard Real-Time Distributed System\*

Matthew Clegg      Keith Marzullo  
University of California, San Diego  
Department of Computer Science and Engineering  
9500 Gilman Drive 0114  
La Jolla, CA 92093-0114  
{mclegg,marzullo}@cs.ucsd.edu

## Abstract

*Processor group membership protocols implement a service that allow processors to agree on which processors are operational. Implementations of group membership for hard real-time systems have concentrated on either reducing failure detection latency or minimizing message complexity. Instead, we present a protocol that uses shared resources—processor time and network bandwidth—as a small, bounded tax imposed on existing broadcast message traffic. In doing so, the group membership protocol can easily be taken into account by any schedulability analysis.*

## 1 Introduction

Group membership is a fundamental service in group-based programming systems. In such systems processes share state both for fault-tolerance and for performance gain through parallelization. These processes keep their shared state coherent by agreeing on a sequence of actions, and by making changes to the shared state through a deterministic protocol that is determined by this sequence of actions. This approach is often referred to as *active replication* or *the state machine approach* [7].

The processor group membership service is the source of the actions that relate to crashes and restarts of the processors; typical actions include notification that a processor has crashed or a processor has restarted. Thus, the group membership service implements a kind of failure detector that allows surviving processes to agree on which processors have failed, which processors are running, and on the order that failures and processor restarts took place.<sup>1</sup>

Conceptually, group membership in a hard real-time system is not a complex service: one builds a failure detector, typically based on a periodic heartbeat message, and then one uses a totally-ordered broadcast protocol to reach agreement on failures and recoveries. More challenging is designing a service that can meet the desired performance requirements. The most common performance requirement that has been considered is a *small failure detection latency*; that is, minimizing the maximum time between when a processor crashes and when the other processors detect the crash (for example, [5] and [6]). This performance requirement is especially important when there is substantial work that has to be done when a failure is detected. The maximum latency has to be included when doing a schedulability analysis, and so a large latency can reduce the schedulability of the system. Another performance requirement that has been considered is *reducing message complexity*, since the number of messages sent can affect the schedulability of the network [5]. Other, more system-specific properties have been examined as well; for example, as well as providing a small failure detection latency, [10] addresses the problem of translating long processor identifiers into short identifiers that can be used as indices into a table.

In the UCSD-CORTO project, we have built a set of protocols that support the development of hard real-time distributed applications. The fault tolerance of this system and the applications that run upon it is based upon active replication. Each critical service is replicated on several processors, and the failure of a small number of processors does not hinder the availability of the service. For our purposes, group membership is required primarily for dynamic reconfiguration, which is a task that can be performed in the background.

Since group membership is not in the critical path, a small failure detection latency is not the most important criteria in our system. Instead, we desire a service that (1) has

\*This research supported by AFOSR Grant F49620-96-1-0215.

<sup>1</sup>Group membership can also be defined at the process level. Such a service is built on top of processor group membership.

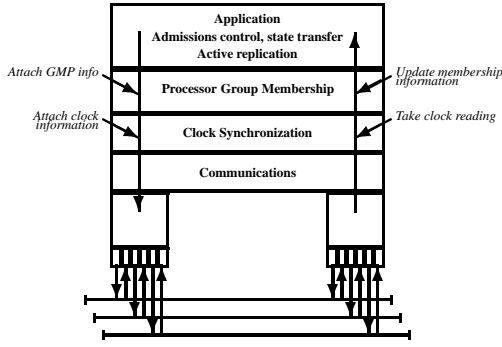


Figure 1. UCSD Corto Architecture

a low overhead, (2) has a low cost of handling failures, and (3) supports a straightforward schedulability analysis.

We provide two different group membership service architectures that allow for its costs to be easily included into a schedulability analysis:

1. *Group membership as a purely reactive service.* In this architecture, the group membership service depends on the applications to generate messages frequently enough to serve as heartbeats. The group membership service obtains its resources as a *small bounded tax* imposed on network bandwidth and on processor computation time. That is, some information is piggy-backed on each message broadcast by the application, and the time needed to deliver a broadcast is increased.
2. *Group membership as a set of periodic processes.* These processes send and receive group membership information and are included in the schedulability analysis of the system.

This second architecture can be factored into two components: a set of periodic processes that send and receive empty messages, and a group membership service built following the first method but that imposes its tax only on this set of periodic processes. Hence, we concentrate on the first method and return to the second method when we discuss the imposed tax.

Figure 1 shows a simple schematic of the UCSD-CORTO architecture. The group membership service is below such services as admissions control, state transfer, and support for active replication, and is above clock synchronization and real-time communications. The group membership service does not generate any messages *per se*, but rather relies on the application to generate the broadcast traffic that will serve as a “heartbeat” for failure detection purposes. The broadcast messages are then passed down to the clock synchronization service [3], which, like the group membership

service, attaches some information onto the messages. On the receiving end of the broadcasts, both the clock synchronization service and the group membership service strip off the information that they need to supply their service.

The UCSD-CORTO architecture is built using redundant broadcast channels. It is very inexpensive to equip a set of processors with redundant channels and it requires only commercial off-the-shelf hardware to do so. Furthermore, using redundant channels allows a system to be able to mask long-lasting channel failures. If nonredundant communication were instead used, then the failure model would need to distinguish intermittent failures from more long-lasting failures, and have isolated processors detect long-lasting failures [1].

We briefly note a few other points of comparison between our protocol and other related protocols that have recently appeared in the literature. Our protocol differs substantially from the RTCAST group membership protocol [1] in its approach to the handling of failures. Our protocol masks all failures except crash failures; by contrast, RTCAST masks no failures whatsoever, and instead translates all failures into processor crashes. The PRHB/ED protocol [6], while similar to ours in its ability to mask failures, nonetheless has a somewhat weaker failure model than ours, in that it assumes no more than one failure per round. Moreover, the PRHB/ED protocol assumes a TDMA bus, whereas our protocol assumes only a real-time local area network with a broadcast facility. Cristian’s protocol [5, 4] does not provide group membership in a manner that makes for a simple schedulability analysis. For example, the persistent failure of a single out-adaptor can cause a significant amount of additional message traffic, while in our protocol, *no* additional message traffic will result, provided that a parameter of our protocol,  $\Delta_{fwd}$ , is chosen appropriately.

The paper proceeds as follows. In Section 2 we give a formal specification of group membership for hard real-time systems and compare it with the most-often referenced specifications for the problem. Section 3 specifies the environment in which the UCSD-CORTO group membership protocol runs. Section 4 presents the UCSD-CORTO group membership protocol and proves it correct. Section 5 describes issues related to the group membership protocol.

## 2 Specification

Given is a set  $N$  of  $n$  processors. Each processor  $p \in N$  can either be *crashed* or *not crashed*. The group membership protocol further divides the *not crashed* state into *restarting* and *running*. These different states are denoted by the predicates **crashed** <sub>$p$</sub> ( $t$ ), **restarting** <sub>$p$</sub> ( $t$ ) and **running** <sub>$p$</sub> ( $t$ ), indicating that processor  $p$  is in the crashed state, in the restarting state, or in the running state at time  $t$

respectively. We use the following shorthand:

$$\begin{aligned}
\mathbf{crashed}_p(t_1, t_2) &\stackrel{\text{def}}{=} \forall t : t_1 \leq t < t_2 : \mathbf{crashed}_p(t) \\
\mathbf{crash}_p(t) &\stackrel{\text{def}}{=} \mathbf{crashed}_p(t) \wedge (t = 0 \vee \\
&\quad (\lim_{e \rightarrow 0} \neg \mathbf{crashed}_p(t - e)) \\
\mathbf{restarting}_p(t_1, t_2) &\stackrel{\text{def}}{=} \forall t : t_1 \leq t < t_2 : \mathbf{restarting}_p(t) \\
\mathbf{restart}_p(t) &\stackrel{\text{def}}{=} \mathbf{restarting}_p(t) \wedge (t = 0 \vee \\
&\quad (\lim_{e \rightarrow 0} \neg \mathbf{restart}_p(t - e)) \\
\mathbf{running}_p(t_1, t_2) &\stackrel{\text{def}}{=} \forall t : t_1 \leq t < t_2 : \mathbf{running}_p(t) \\
\mathbf{run}_p(t) &\stackrel{\text{def}}{=} \mathbf{running}_p(t) \wedge (t = 0 \vee \\
&\quad (\lim_{e \rightarrow 0} \neg \mathbf{running}_p(t - e))
\end{aligned}$$

We assume that the state of a processor is piecewise constant, i.e., that in any finite period of time, a processor will change states at most a finite number of times. This ensures that the above limits are meaningful. Each processor is initially restarting or crashed, and the protocol moves processors from the *restarting* state to the *running* state. The environment, however, can cause a transition to the *crashed* state at any time. Hence, we have:

**GMP0** : *Valid States*

$$\begin{aligned}
&\forall p \in N, \forall t \geq 0 : \\
&\quad (\mathbf{crashed}_p(t) \wedge \neg \mathbf{restarting}_p(t) \wedge \neg \mathbf{running}_p(t) \vee \\
&\quad \neg \mathbf{crashed}_p(t) \wedge \mathbf{restarting}_p(t) \wedge \neg \mathbf{running}_p(t) \vee \\
&\quad \neg \mathbf{crashed}_p(t) \wedge \neg \mathbf{restarting}_p(t) \wedge \mathbf{running}_p(t)) \\
&\wedge \neg \mathbf{running}_p(0) \\
&\wedge \mathbf{crashed}_p(t) \Rightarrow \lim_{e \rightarrow 0} (\mathbf{crashed}_p(t + e) \vee \\
&\quad \mathbf{restart}_p(t + e)) \\
&\wedge \mathbf{running}_p(t) \Rightarrow \lim_{e \rightarrow 0} (\mathbf{running}_p(t + e) \vee \\
&\quad \mathbf{crash}_p(t + e))
\end{aligned}$$

Each processor  $p$  has a physical clock  $C_p$  that reports the time as a discrete value. Without loss of generality, we can let the domain of real time be the nonnegative real numbers and the range of clock values be the nonnegative integers. We assume that the clocks of the processors are synchronized within  $\epsilon$  of each other, that the drift of the clocks is small enough to ignore, and that the clocks have a fine enough granularity such that each process obtains a unique value each time it reads its clock. The last two assumptions are easy to remove; we assume them to simplify the presentation.

The group membership protocol implements, for each processor, a set  $\mathbf{Members}_p(T)$  of processor identifiers, which represents  $p$ 's view of the membership when  $C_p$  had the value  $T$ . This is called the *membership set* of  $p$ . Informally, the specification of group membership has two run-

ning processors agree on the values of their membership sets when the clocks on the two processors have the same value.

We list below the properties which specify a real-time group membership protocol. The first two properties state that the running processors agree amongst themselves that they are indeed running, the third property bounds how long a processor can be crashed yet still be in the membership, and the last property bounds on how long a processor can be in the restarting state.

**GMP1** : *Reflexivity*

$$\begin{aligned}
&\forall p \in N, \forall t : \mathbf{running}_p(t) \Rightarrow \\
&\quad p \in \mathbf{Members}_p(C_p(t))
\end{aligned}$$

**GMP2** : *Agreement*

$$\begin{aligned}
&\forall p, q \in N, \forall t_1, t_2 : \\
&\quad C_p(t_1) = C_q(t_2) \wedge \mathbf{running}_p(t_1) \wedge \mathbf{running}_q(t_2) \Rightarrow \\
&\quad \mathbf{Members}_p(C_p(t_1)) = \mathbf{Members}_q(C_q(t_2))
\end{aligned}$$

**GMP3** : *Crash Completeness*

$$\begin{aligned}
&\exists \Delta_{lat} : \forall p, q \in N, \forall t : \\
&\quad \mathbf{crashed}_p(t) \wedge \mathbf{running}_q(t + \Delta_{lat}) \Rightarrow \\
&\quad p \notin \mathbf{Members}_q(t + \Delta_{lat})
\end{aligned}$$

**GMP4** : *Bounded Restarting*

$$\begin{aligned}
&\exists \Delta_{rub}, \Delta_{rlb} : \forall p \in N, \forall t : \\
&\quad \neg \mathbf{restarting}_p(t, t + \Delta_{rub}) \\
&\wedge \mathbf{running}_p(t + \Delta_{rlb}) \Rightarrow \\
&\quad \forall t' : t \leq t' < t + \Delta_{rlb} : \neg \mathbf{crashed}_p(t')
\end{aligned}$$

Thus,  $\Delta_{lat}$  is the maximum failure detection latency,  $\Delta_{rub}$  is an upper bound on how long a processor can be in the restarting state, and  $\Delta_{rlb}$  is a lower bound on how long a processor can be in the restarting state. To see the last point, consider a time  $t_p$  at which  $\mathbf{run}_p(t_p)$  holds. From **GMP0** the state of  $p$  immediately preceding  $t_p$  must be the restarting state, and so **GMP4** implies that the restarting state must have endured at least  $\Delta_{rlb}$ .

An immediate consequence of **GMP1**–**GMP4** is that  $\Delta_{rlb} > \Delta_{lat} + \epsilon$ . To see this, consider processors  $p$  and  $q$  such that  $p$  is crashed at time  $t_0$ ,  $q$  is running at  $t_0$ , and  $q$  does not crash. According to **GMP4**,  $p$  could re-enter the running state as early as  $t_0 + \Delta_{rlb}$ , which from **GMP1** and **GMP2** implies that  $p \in \mathbf{Members}_q(C_p(t_0 + \Delta_{rlb}))$ . But, according to **GMP3**,  $p \notin \mathbf{Members}_q(C_q(t_0 + \Delta_{lat}))$ . Since only the environment can cause a processor to crash, we must have  $C_q(t_0 + \Delta_{lat}) < C_p(t_0 + \Delta_{rlb})$ . From the assumption that the clocks are synchronized within  $\epsilon$  of each other,  $C_q(t_0 + \Delta_{rlb}) - \epsilon \leq C_p(t_0 + \Delta_{rlb})$ , and so we must

have  $C_q(t_0 + \Delta_{lat}) < C_q(t_0 + \Delta_{rlb}) - \epsilon$ .  $\Delta_{rlb} > \Delta_{lat} + \epsilon$  follows from the assumption that the drift of the clocks is small enough to ignore.

Other specifications for hard real-time group membership have been informal, and so it is hard to compare them with ours. The specification given in [6] can easily be accepted as an informal version of ours. It is harder to see such a correspondence with the earliest specification for the problem, given by Cristian in [5]. There are superficial differences, the most obvious being that his specification is in terms of processors joining and leaving groups. It is not hard to build a refinement mapping that allows one to rewrite our specification in terms of processor groups. Once such superficial differences are addressed, it is easy to see that our specification rejects behaviors that satisfies Cristian's. For example, in his specification two running processors are not constrained to agree on the membership based on the values of their clocks, while they are in ours (and in the specification in [6]). Furthermore, one reasonable interpretation of his specification rejects behaviors that satisfies ours. Property *Sa* in [5] states:

After a processor joins a group, it stays joined to that group until a failure is detected or a processor start occurs.

More formally, if, for two times  $t_1$  and  $t_2 > t_1$ ,  $\mathbf{Members}_p(C_p(t_1)) \neq \mathbf{Members}_p(C_p(t_2))$ , then there must have been a processor that either restarted or whose failure was detected at some time  $t : t_1 \leq t \leq t_2$ . This need not be the case for our specification. For example, consider a time  $t_0$  at which processor  $p$  is running. At time  $t_q > t_0$ ,  $q$  starts, and then at time  $t_r > t_q$ ,  $r$  starts. Let  $t_1$  be the time that  $p$  adds  $q$  to its membership set and  $t_2$  the time  $p$  adds  $r$  to its membership set. If  $t_r - t_q < \Delta_{rlb}$  then it is possible for  $t_r < t_1 < t_2$  thereby violating *Sa*. Thus, it can be argued that our specification is incomparable to the one in [5].<sup>2</sup>

One behavior that our specification admits and that one may find troubling is the following: consider a running processor  $p$  and a restarting processor  $q$ . It is legal according to our specification for  $p$  to add  $q$  to its membership set at some clock time  $T$  without  $q \in \mathbf{Members}_q(T)$ . We do not consider this a problem, however, because either  $q$  will eventually crash, or  $q$  will enter the running state (Property **GMP4**) at which time  $p$  and  $q$  will agree on the membership (Property **GMP2**). We don't know whether past work in the area has considered this behavior a problem or not; this is

<sup>2</sup>Cristian has told us in private correspondence that his specification was informal in order to make it acceptable to engineers working on the IBM/FAA Air Traffic Control System project. Unfortunately, the informal nature of the specification makes it ambiguous, and so we cannot tell whether the service he had in mind is indeed incomparable to the one we specify here.

the kind of subtle detail that is often overlooked when writing informal specifications.

### 3 System Model

We assume that a crash has a minimum duration  $\Delta_{crash}$  that is at least as long as  $\Delta_{lat}$ :

**GMP5** : *Crash Duration*

$$\exists \Delta_{crash} \geq \Delta_{lat} : \mathbf{crash}_p(t) \Rightarrow \mathbf{crashed}_p(t, t + \Delta_{crash})$$

**GMP5** is not a difficult property to implement. When a processor recovers from a crash, it typically executes a large amount of code before finally initializing the group membership service. This period of time, from being rebooted to initializing group membership, can be considered the *crashed* state by the group membership service, and is usually much longer than the maximum failure detection latency. If it is not, however, then the group membership initialization routine can first sleep for as long as necessary to ensure that **GMP5** holds.

We assume that the processors are connected by  $c_{max}$  channels, which is an idealized broadcast bus. Each processor has an in-adapter and an out-adapter for each of the  $c_{max}$  channels. The  $send_p(m, c)$  event places the unique message  $m$  on processor's  $p$  out adapter. The channel connected to that out-adapter then takes that message and presents it to all of the in-adapters connected to channel  $c$ . The in-adapters then present message  $m$  to the processor through the execution of a  $receive_p(m, c)$  event. The time between the  $send_p(m, c)$  event and the corresponding  $receive_q(m, c)$  event is bounded from above by  $\Delta_{send}$ .

Each processor  $p$  is guaranteed to execute a  $gmp\_send_p(m)$  event for some unique message  $m$  at least once every  $\delta > \epsilon$  seconds and no more frequently than once every  $\Delta_{send}$  seconds.

The following failures can occur:

- Up to  $f_{crash}$  processors can be in the crashed state at any time.
- Up to  $f_{out}$  out-adapters can be faulty. A faulty out-adapter can silently drop any message given to it by its processor rather than presenting it to the channel to which the out-adapter is connected.
- Up to  $f_{in}$  in-adapters can be faulty. A faulty in-adapter can either silently drop any message that is on the channel to which it is connected rather than presenting it to its processor, or it can deliver the message later than  $\Delta_{send}$  after the message was presented to the original out-adapter.
- Up to  $f_{chan}$  channels can be faulty. A faulty channel can silently drop any message given to it by a con-

nected out-adapter rather than presenting it to all of the connected in-adapters.

**Theorem 1** *If  $f_{in} + f_{out} + f_{chan} < c_{max}$  and a process  $p$  executes  $send_p(m, c)$  for all channels  $1, 2, \dots, c_{max}$  by time  $t$ , then all non-crashed processes will deliver  $m$  by  $t + \Delta_{send}$ .*

**Proof:** A proof of this can be found in [2].  $\square$

## 4 Protocol

Our protocol is based on agreement protocols for systems that have redundant broadcast channels [2, 4]. Rather than actively sending messages, however, the protocol relies on the layers of the system above group membership to generate sufficiently frequent messages to both make progress in the agreement protocol and to serve as a “heartbeat”; if a processor does not broadcast after a certain time, then that processor is detected as having crashed.

We assume that  $n > f_{crash} + f_{in}$  and  $f_{in} + f_{out} + f_{chan} < c_{max}$ .

### 4.1 Pseudocode

Figure 4.1 shows a pseudocode version of the group membership protocol. In this figure, **me** is the identity of the processor running the protocol, and **now()** is the current value of the processor’s clock. The other constants in the protocol are as follows:

- $\delta$ , the maximum time between successive broadcasts by any non-crashed processor. Any broadcast of a processor will invoke the routine *gmp\_send*. In Section 5, we consider the problem of dynamically changing the value of  $\delta$ .
- $\epsilon$ , the maximum clock skew, is a constant whose value is defined by the clock synchronization service.
- $\Delta_{send}$ , the maximum one-way message transmission delay, is defined by the network communications layer.
- $\Delta_{crash}$ , the minimum time that a processor is in the *crashed* state, has a value that is bounded from below by the protocol. For this protocol,  $\Delta_{crash}$  must be at least  $\Delta_{send} + \Delta_{sf} + 2\delta + 2\epsilon$ ,
- $\Delta_{fwd}$ , a protocol constant that is used to determine when to relay a heartbeat onto another channel. We consider appropriate values of  $\Delta_{fwd}$  in Section 5.
- $\Delta_{sf}$  is the larger of  $\Delta_{send}$  and  $\Delta_{fwd}$ .

The three global variables in the protocol are:

```

array [1..n] of Time recv;
array [1..n] of Time join;
array [1..n] of Channel chan;

const  $\Delta_{sf} = \max(\Delta_{fwd}, \Delta_{send});$ 

void gmp_restart() {
    Time T = now();
    for (i = 1; i  $\leq$  n; i++) {
        recv[i] =  $-\infty$ ; join[i] =  $\infty$ ; }
    recv[me] = T;
    join[me] = T +  $\Delta_{send} + \Delta_{sf} + 3\delta + 2\epsilon$ ;
}

int gmp_member(Processor i) {
    Time T = now();
    return join[i]  $\leq$  T && T < recv[i] +  $\Delta_{send}$ 
        +  $\Delta_{sf} + 2\delta + \epsilon$ ;
}

int gmp_restarting() { return ! gmp_member(me); }

int gmp_running() { return gmp_member(me); }

void gmp_send() {
    recv[me] = T = now();
    for (c = 1; c  $\leq$   $c_{max}$ ; c++) gmp_csend(c, T); }

void gmp_csend(Channel c, Time T) {
    send([me, T], c);
    for (i = 1; i  $\leq$  n; i++)
        if (i  $\neq$  me && chan[i] < c
            && recv[i] +  $\Delta_{sf} <$  T) {
            send([i, recv[i]], c); chan[i] = c; }
}

void gmp_rcv(Channel c, Processor i, Time T) {
    if (recv[i]  $\leq$  now() -  $\Delta_{send} - \Delta_{sf} - 2\delta - \epsilon$ )
        join[i] = T +  $\Delta_{send} + \Delta_{sf} + 2\delta + \epsilon$ ;
    if (recv[i] < T) { recv[i] = T; chan[i] = c; }
    else if (recv[i] == T && chan[i] < c) chan[i] = c;
}

```

**Figure 2. Group Membership protocol**

- *recv*, used to implement the heartbeat. The value of  $\text{recv}[i]$  is the latest clock value that processor **me** knows that processor  $i$  sent.
- *join*, used to agree on when a processor has restarted. When **now**() is at least  $\text{join}[i]$ , then  $i$  has joined the group of operational processors.
- *chan*, used to implement the reliable broadcast. The value of  $\text{chan}[i]$  is the highest channel upon which **me** knows that  $\text{recv}[i]$  was sent.

## 4.2 Proof

We say that “ $p$  sends  $T$  at  $t$ ” to mean that processor  $p$  called *gmp\_send* at time  $t$  such that  $C_p(t) = T$ , and we say that “ $p$  sends  $T$  at  $t$  without crashing” to mean that  $p$  sent  $T$  at  $t$  and does not crash before *gmp\_send* terminates. We use the notation  $T_p^i$  to indicate the clock time that processor  $p$  had when it called *gmp\_send* for the  $i^{\text{th}}$  time since it last left the *restarting* state.

We define the condition  $\text{crashed}_{\text{me}}(t)$  to hold when **me** is crashed at  $t$  or **me** has restarted from a crash but has not yet started the group membership service. We define  $\text{running}_{\text{me}}(t)$  to hold when  $\neg \text{crashed}_{\text{me}}(t) \wedge \text{join}[\text{me}] \leq C_{\text{me}}(t) < \text{recv}[\text{me}] + \Delta_{\text{send}} + \Delta_{\text{sf}} + 2\delta + \epsilon$ , and define  $\text{restarting}_{\text{me}}(t)$  to hold when  $\neg \text{crashed}_{\text{me}}(t) \wedge \neg \text{running}_{\text{me}}(t)$  holds. This implies that in the initial state of the system, all of the processors are in the *crashed* state. Finally, once  $p$  is in the *running* state, it will, by the periodicity of broadcasts assumption, set  $\text{recv}[p]$  to **now**() every  $\delta$ , and so  $0 \leq C_{\text{me}}(t) - \text{recv}[p] \leq \delta$  which satisfies  $\text{running}_p(t)$ . Thus, **GMP0** holds.

**Theorem 2** *If  $p$  sent  $T$  at  $t$  without crashing and  $q$  does not crash during the interval of time  $[t, t + \Delta_{\text{send}}]$ , then  $q$  sets  $\text{recv}[p]$  to  $T$  during the interval of time  $[t, t + \Delta_{\text{send}}]$ .*

**Proof:** From the definition of  $\Delta_{\text{send}}$  and Theorem 1.  $\square$

**Theorem 3** *If  $p$  sends  $T_1$  and then sends  $T_2$ , then any processor that sets  $\text{recv}[p]$  to  $T_1$  and sets  $\text{recv}[p]$  to  $T_2$  does so in this order.*

**Proof:** From Theorem 2 and the fact that  $p$  does not send more often than one message every  $\Delta_{\text{send}}$ .  $\square$

### Theorem 4

*If  $p$  sent  $T$  at  $t$ ,  $\text{running}_q(t, t + \Delta_{\text{send}} + \Delta_{\text{sf}} + \delta + \epsilon)$ , and  $\text{running}_r(t, t + \Delta_{\text{send}} + \Delta_{\text{sf}} + \delta + \epsilon)$ , then  $q$  sets  $\text{recv}[p]$  to  $T$  during the interval of time  $[t, t + \Delta_{\text{send}} + \Delta_{\text{sf}} + \delta + \epsilon]$  iff  $r$  sets  $\text{recv}[p]$  to  $T$  during the interval of time  $[t, t + \Delta_{\text{send}} + \Delta_{\text{sf}} + \delta + \epsilon]$ .*

**Proof:** Suppose  $p$  sends  $T$  at  $t$  and  $q$  sets  $\text{recv}[p]$  to  $T$  during the interval of time  $[t, t + \Delta_{\text{send}} + \Delta_{\text{sf}} + \delta]$ . Since  $q$  did so, it follows that some processor (possibly one that subsequently crashed), received  $T$  in the time interval  $[t, t + \Delta_{\text{send}}]$ . Let  $s$  be the processor that first received  $T$  and let  $c$  be a channel on which  $s$  received  $T$ . It follows that neither  $p$ 's out-adapter for channel  $c$  nor channel  $c$  itself failed during the transmission of  $T$ . Since there are at least  $f_{\text{crash}} + 1$  processors connected to channel  $c$ , it further follows there must have been at least one processor  $x$  that was running in  $[t, t + \Delta_{\text{send}} + \Delta_{\text{sf}} + \delta]$  and whose in-adapter did not fail. Therefore,  $x$  must have received  $T$  in the interval  $[t, t + \Delta_{\text{send}}]$ .

Let  $t_0$  be the first time after  $t + \Delta_{\text{sf}} + \epsilon$  at which  $x$  executes *gmp\_send*. By assumption of the periodicity of message transmission, we have  $t_0 \leq t + \Delta_{\text{sf}} + \delta + \epsilon$ , and from above we know that  $x$  has received  $T$  from  $p$  before  $t_0$ .

Let  $T' = C_x(t_0)$ . Suppose  $\text{recv}[p] \neq T$  at  $x$  at time  $t_0$ . Since  $x$  delivered  $T$  before  $t_0$ , from Theorem 3 it follows that  $T < \text{recv}[p]$ . By assumption,  $\Delta_{\text{crash}} \geq \Delta_{\text{send}} + \Delta_{\text{sf}} + 3\delta + \epsilon > \Delta_{\text{sf}} + \delta + \epsilon$ . Hence, from **GMP5** it follows that  $p$  could not have crashed between the time that it transmitted  $T$  on channel  $c$  and when it transmitted the message containing timestamp  $T'$ . Thus,  $p$  must have sent  $T$  at  $t$  without crashing, and the theorem follows from Theorem 2.

Otherwise, we may assume that  $\text{recv}[p] = T$  at  $x$  at time  $t_0$ . Let  $c_x$  be the highest channel on which  $x$  received  $T$  by time  $t_0$ . If  $c_x > f_{\text{chan}} + f_{\text{in}} + f_{\text{out}}$ , then  $p$  sent  $T$  on at least  $f_{\text{chan}} + f_{\text{in}} + f_{\text{out}} + 1$  channels. It follows from the failure model assumption that there must have been one channel  $c$  such that  $c$  did not fail, the out-adapter for  $p$  did not fail, and the in-adapter for  $r$  did not fail. Therefore,  $r$  must have received  $T$  by  $t + \Delta_{\text{send}}$  and the theorem follows.

Otherwise, suppose that  $c_x < f_{\text{chan}} + f_{\text{in}} + f_{\text{out}} + 1$ . Therefore, during the execution of *gmp\_send* at time  $t_0$  by  $x$ , the message  $[p, T]$  is transmitted on each of the channels  $c_x + 1 \dots c_{\text{max}}$ . Since  $c_{\text{max}} > f_{\text{chan}} + f_{\text{in}} + f_{\text{out}}$ , there must be at least one channel  $c$  such that  $c$  did not fail,  $s$ 's out-adapter for  $c$  did not fail and  $r$ 's in-adapter for  $c$  did not fail. Therefore,  $r$  must have received  $[p, T]$  by  $t_0 + \Delta_{\text{send}} \leq t + \Delta_{\text{sf}} + \Delta_{\text{send}} + \delta + \epsilon$ .  $\square$

**Theorem 5** *The protocol implements **GMP1**.*

**Proof:** Directly from *gmp\_running* and definition of  $\text{running}_{\text{me}}(t)$ .  $\square$

**Theorem 6** *The protocol implements **GMP4** for  $\Delta_{\text{rub}} = \Delta_{\text{send}} + \Delta_{\text{sf}} + 4\delta + 3\epsilon$  and  $\Delta_{\text{rlb}} = \Delta_{\text{send}} + \Delta_{\text{sf}} + 3\delta + 2\epsilon$ .*

**Proof:** From **GMP1** and the fact that a restarting process that does not crash will make its first broadcast within  $\delta$  of entering the *restarting* state.  $\square$

**Theorem 7** *The protocol implements GMP3 for  $\Delta_{lat} = \Delta_{send} + \Delta_{sf} + 2(\delta + \epsilon)$ .*

**Proof:** Assume that processor  $p$  crashes at time  $t_0$ , and the last timestamp that  $p$  sent before crashing was  $T_p^k$ ; hence,  $T_p^k < C_p(t_0)$ . From the protocol, the earliest time at which  $\mathbf{Members}_q(t)$  will no longer contain  $p$  is when:

$$C_{\mathbf{me}}(t) - \Delta_{send} - \Delta_{sf} - 2\delta - \epsilon = T_p^k \quad (1)$$

Starting from Equation 1,

$$\begin{aligned} C_{\mathbf{me}}(t) - T_p^k &= \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon \\ C_{\mathbf{me}}(t) - C_p(t_0) &< \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon \\ t - t_0 + C_{\mathbf{me}}(t_0) - C_p(t_0) &< \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon \\ t - t_0 &< \Delta_{send} + \Delta_{sf} + 2\delta + 2\epsilon \quad (2) \end{aligned}$$

Thus,  $q$  will remove  $p$  from  $\mathbf{Members}_q$  within  $\Delta_{send} + \Delta_{sf} + 2\delta + 2\epsilon$  of  $p$ 's crash. From **GMP5**, we then conclude  $\mathbf{crash}_p(t) \wedge \mathbf{running}_q(t + \Delta_{lat}) \Rightarrow p \notin \mathbf{Members}_q(t + \Delta_{lat})$  for  $\Delta_{lat} = \Delta_{send} + \Delta_{sf} + 2\delta + 2\epsilon$ . The theorem follows from Theorem 6. Note that since we constrain  $\delta > \epsilon$ ,  $\Delta_{rlb} > \Delta_{lat} + \epsilon$  which is required for **GMP1–GMP4** to be consistent.  $\square$

**Theorem 8** *The protocol implements GMP2.*

**Proof:** Consider two processors  $p$  and  $q$ , both running at times  $t_1$  and  $t_2$  respectively, and  $C_p(t_1) = C_q(t_2) = T$  where there is some processor  $r$  that is in  $\mathbf{Members}_p(T)$  and not in  $\mathbf{Members}_q(T)$ . Since  $r \in \mathbf{Members}_p(T)$  and  $r \notin \mathbf{Members}_q(T)$ , there are four (not necessarily distinct) timestamps from  $r$  such that:

$$\begin{aligned} T_r^h + \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon &\leq T \quad \wedge \\ T_r^i + \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon &> T \end{aligned}$$

$$\begin{aligned} T_r^j + \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon &> T \quad \vee \\ T_r^k + \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon &\leq T \end{aligned}$$

where  $T_r^\ell = C_r(t_r^\ell)$  for  $\ell$  equal to  $h, i, j$  or  $k$ . There are two cases:

1.  $T_r^j + \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon > T$ . It immediately follows that  $T_r^j > T_r^h$ , meaning that processor  $q$  has a later value for  $\text{join}[r]$  than processor  $p$  has. The original inequality can be rewritten as follows:

$$\begin{aligned} C_q(t_2) - C_r(t_r^j) &< \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon \\ t_2 - t_r^j &< \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon + C_r(t_r^j) - C_q(t_r^j) \\ t_2 - t_r^j &< \Delta_{send} + \Delta_{sf} + 2\delta + 2\epsilon \quad (3) \end{aligned}$$

If  $t_q^0$  denotes the time that processor  $q$  restarted, then we know by the periodicity of broadcasts that  $t_r^j - t_q^0 \leq \delta$ . Combining with Equation 3, we get  $t_2 - t_q^0 < \Delta_{send} + \Delta_{fwd} + 3\delta + 2\epsilon$ . Since this is less than  $\Delta_{rlb}$ , process  $q$  is not running at  $t_2$ , a contradiction.

2.  $T \geq T_r^k + \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon$ . It immediately follows that  $T_r^i > T_r^k$ ; that is, processor  $p$  has received  $T_r^i$  while processor  $q$  has not yet received  $T_r^i$ . From Theorem 4, this implies that  $t_2 - t_r^i < \Delta_{send} + \Delta_{sf} + \delta$ . Furthermore, by Theorem 3,  $i = k + 1$  and so  $T_p^i = T_p^{k+1}$ . Rewriting the inequality, we get

$$\begin{aligned} C_q(t_2) - C_r(t_r^k) &\geq \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon \\ t_2 - t_r^k &\geq \Delta_{send} + \Delta_{sf} + 2\delta + \epsilon + C_r(t_r^k) - C_q(t_r^k) \\ t_2 - t_r^k &\geq \Delta_{send} + \Delta_{sf} + 2\delta \quad (4) \end{aligned}$$

Combining Equation 4 with  $t_2 - t_r^{k+1} < \Delta_{send} + \Delta_{fwd} + \delta$ , we get  $t_r^{k+1} - t_r^k > \delta$ , which violates the assumption that broadcasts occur at least once every  $\delta$ .  $\square$

## 5 Discussion

In this section, we consider some details of the group membership service implementation.

### 5.1 Forwarding

In our distributed testbed [3],  $\Delta_{send} = \Delta_{fwd} = 2$  msec,  $\delta \leq 40$  msec, and  $\epsilon = 1$  msec. This gives  $\Delta_{lat} = 86$  msec,  $\Delta_{rlb} = 126$  msec, and  $\Delta_{rub} = 166$  msec. If instead a protocol like [5] or [2] were used to agree on a sequence of failure detections based on a  $\delta$ -frequent heartbeat, then  $\Delta_{lat}$  could be as small as  $\delta + 2\Delta_{send} + \epsilon = 45$  msec.

As well as not providing group membership as a fixed tax, a drawback of the latter approach is the message complexity when an out-adapter or a channel failure occurs. The failure is observed by all of the processors within the tightness of the network [9], and so a large fraction of the processors will redundantly forward the dropped group membership information. In our protocol, the times that the different processors send their broadcast messages is presumably spread out for schedulability purposes, and so such redundant forwarding is much less likely to occur.

If desired, one can further reduce the expected amount of forwarding of group membership information by increasing  $\Delta_{fwd}$ . Clearly, it makes no sense to have  $\Delta_{fwd}$  less than  $\Delta_{send}$ , since doing so does not improve the maximum failure detection latency and may cause information to be

forwarded even when there were no failures. If  $\Delta_{fwd} \geq \delta + \Delta_{send} + \epsilon$ , then the only time that information would be forwarded would be if a processor were to crash while in *gmp\_send*. Any failures of channels or link adaptors (which one might expect to be much more likely to occur than crash failures) will not require any information to be forwarded.

## 5.2 Tax

There are two taxes that this group membership protocol imposes: one on execution time and one on message size. Both the execution time tax and the message size tax includes the cost of copying and carrying the group membership information pairs of a processor identifier and a timestamp.

The execution tax is the additional time imposed by the group membership protocol on the execution of the application's  $send_p(m)$  and corresponding  $receive_q(m)$  invocations. The overhead with  $gmp\_send_p(m)$ , invoked as a result of  $send_p(m)$ , is the most complex to compute, since it depends on the failures that may have occurred in the recent past. If we assume that no more than  $f$  failures can occur between any two invocations of *gmp\_send*, then the total number of copied into messages generated by *gmp\_send* is bounded by at most  $fc_{max}$ . Therefore, the running time of *gmp\_send* is bounded by at most  $C_0 + C_1fc_{max}$  for some constants  $C_0$  and  $C_1$ .

Each message sent by *gmp\_send* can carry up to  $n$  group membership information pairs. The size of such a pair can be reduced by using as few bits as possible for the processor identifiers and the timestamp values. Once the difference between two timestamps is larger than  $\Delta_{send} + \Delta_{sf} + 2\delta + \epsilon$  it no longer matters how far apart they are, and so a representation of  $-\infty$  for timestamps that are older than this would suffice. If we assume that, as part of *gmp\_send*, old timestamps are set to  $-\infty$ , then it suffices to have enough bits to represent the range  $2(\Delta_{send} + \Delta_{sf} + 2\delta + \epsilon)$  plus a bit to represent  $-\infty$ . If the granularity of the clock is  $\gamma$ , then this requires  $\log_2((\Delta_{send} + \Delta_{sf} + 2\delta + \epsilon/\gamma) + 2)$  bits. For our system,  $\gamma$  is 1  $\mu$ sec and  $n = 4$ , and so the piggyback size can be represented in as few as 11 bytes.<sup>3</sup>

## 5.3 Periodic Process Model

Rather than piggybacking information on the application's messages, an alternate approach is to implement periodic processes that execute *gmp\_send* and execute *gmp\_recv* for all of the channels. The application schedulability analysis would then include these periodic processes.

<sup>3</sup>Because the underlying clock synchronization protocol uses some of the same information that the group membership protocol uses [3], such a severe compression would be counterproductive. In particular, the timestamp of the processor that is broadcasting should be a full representation.

Let  $\rho_S$  be the period of the *gmp\_send* process and  $\rho_R$  be the period of the *gmp\_recv* process. Note that we must have  $\rho_S \geq \rho_R(n - 1)$ , or else the volume of messages produced by invocations of *gmp\_send* on remote processors will exceed the capability of *gmp\_recv* to consume them.

To compute the end-to-end delay from when *gmp\_send* is invoked until the message is processed by a corresponding invocation of *gmp\_recv*, we assume that the network is scheduled using the synchronous streams model [8]. If the periodicity of message transmission is chosen to be equal to the periodicity of execution of *gmp\_send*, then the end-to-end delay of message transmission can be bounded by  $C_{send} + 2\rho_S$  for some constant  $C_{send}$ . When a message arrives in an in-adaptor, there may be a queue of up to  $n - 1$  other messages from invocations of *gmp\_send* on other processors. Consequently, a bound on the time before a message will be delivered and processed by an invocation of *gmp\_recv* is  $(n + 1)\rho_R \leq \frac{n+1}{n-1}\rho_S$ .

Thus, a bound on  $\Delta_{send}$  is given by summing the time needed to execute *gmp\_send*, the time needed to transmit the messages generated by *gmp\_send*, and the time needed to executing the corresponding invocations of *gmp\_recv*. Thus,  $\Delta_{send} = \rho_S + 2\rho_S + C_{send} + \frac{n+1}{n-1}\rho_S = (3 + \frac{n+1}{n-1})\rho_S + C_{send}$ .

Since the elapsed time between two successive invocations of *gmp\_send* may be up to  $2\rho_S$ , we must take  $\delta \geq 2\rho_S$ . This yields a value of  $\Delta_{lat} \geq (10 + \frac{2(n+1)}{n-1})\rho_S + 2C_{send} + 2\epsilon$ .

## 5.4 Mode Changes

The value of  $\delta$  (and indirectly,  $\Delta_{send}$  whose value may depend on  $\delta$ ) is defined by the application rather than by the environment. Different operational modes of the application may result in different values of  $\delta$ , and so the group membership service must be kept abreast of such changes. We assume that the application decides that, at a given clock time  $T$ , the value of  $\delta$  is to be changed to  $\delta'$ . There are two cases to consider:

1.  $\delta' < \delta$ . At clock time  $T + \delta' + \epsilon$  each processor knows that the other running processors will have broadcast a message using the new maximum period  $\delta'$ , which, according to Theorem 4 will be delivered by all running processors within another  $\Delta_{send} + \Delta_{sf} + \delta' + \epsilon$ . Since  $\delta' < \delta$ , all processors that were excluded from the group membership using the old value of  $\delta$  will still be excluded. Hence, the group membership service can simply adopt  $\delta'$  when  $C_{me} = T + \Delta_{send} + \Delta_{sf} + 2\delta' + 2\epsilon$ .
2.  $\delta' > \delta$ . The new value of  $\delta'$  cannot simply be adopted, because doing so could mistakenly reintroduce crashed processors into the group membership. Instead, at time  $T$  each processor can determine the set of processors that are not in **me**'s group membership. For each such processor  $q$ ,  $p$  can set `join[q]`

$\infty$  and  $\text{rcv}[p]$  to  $-\infty$ . Once this is done, the service can adopt the new value  $\delta$ .

## 5.5 Implementation

We have implemented this group membership service on a set of four 486-based PCs running LynxOS. The service is built on top of the UCSD-CORTO clock synchronization service [3] and a communication layer built using a “leaky bucket” abstraction [11] for access control. The processors are each connected to two ethernet, which means that  $f_{chan} + f_{in} + f_{out} \leq 1$ .

Our protocol is based on the reliable broadcast for redundant channels protocol of [4]. A protocol that is very similar to this reliable broadcast protocol is presented in [2]. The most significant difference is in the forwarding rule, represented in our protocol in the implementation of *gmp\_csend*. Like [4], our protocol makes the assumption that if the protocol run by processor  $p$  contains the sequence of instructions

$\dots; \text{send}_p(m, c); \dots \text{send}_p(m', c');$

then if another processor  $q$  delivers  $m'$  but not  $m$ , then there was a failure of at least one of the components (out-adapter of  $p$  for  $c$ , channel  $c$ , in-adapter of  $q$  for  $c$ ). The protocol in [2], however, admits the possibility of a crash of  $p$  causing the same behavior. That is, this sequence of send instructions does not ensure that  $m$  will be consumed by channel  $c$  before message  $m'$  is consumed by channel  $c'$ . If we were to make the same assumption, then  $\text{chan}[i]$  would need to be the set of channels upon which **me** knows that  $\text{rcv}[i]$  was sent, and *gmp\_csend* would be replaced with:

```
void gmp_csend(Channel c, Time T) {
    send([me, T], c);
    for (i = 1; i ≤ n; i++)
        if (i != me && c ∉ chan[i]
            && rcv[q] + Δsf < T) {
            send([i, rcv[i]], c); chan[i] = chan[i] ∪ {c};
        }
}
```

The choice of which implementation of *gmp\_csend* to use depends on the operating system and the underlying machine architecture. For example, on a uniprocessor running most versions of Unix, the original version is a reasonable implementation. The socket implementation serializes all socket-level communication, and so it would take a very unlikely combination of large backoff on one channel, an in-adapter failure on another channel, and a well-timed crash for the old version of *gmp\_csend* to fail. On the other hand, the new version of *gmp\_csend* imposes the same tax as the old version, and does not forward any group membership information if there are no failures. Furthermore, if one uses the larger value of  $\Delta_{fwd}$  described in Section 5.1, the

only time that more information would be forwarded would be if a processor were to crash in the midst of performing *gmp\_send*.

**Acknowledgments** As well as the two authors, the UCSD-CORTO group membership protocol has been worked on by Eric Berry, Walfredo Cirne, Richard Gallup, and Paula Obendorf. The current implementation to Richard Gallup and Paula Obendorf.

## References

- [1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RT-CAST: Lightweight multicast for real-time process groups. In *Second IEEE Real-Time Technology and Applications Symposium*, pages 250–259, Brookline, Massachusetts, June 10-12 1996.
- [2] Ö. Babaoğlu and R. Drummond. Streets of Byzantium: networks architectures for fast reliable broadcasts. *IEEE Transactions on Software Engineering*, 11(6):546–554, June 1985.
- [3] M. Clegg and K. Marzullo. Clock synchronization in hard real-time distributed systems. Technical Report CS96-478, University of California, San Diego Department of Computer Science and Engineering, 1996.
- [4] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2(3):195–212, September 1990.
- [5] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, 1991.
- [6] K. Kim et al. An efficient decentralized approach to processor-group membership maintenance in real-time LAN systems: the PRHB/ED scheme. In *Eleventh Symposium on Reliable Distributed Systems*, pages 74–83, Houston, Texas, October 5-7 1992.
- [7] K. Krithi, K. Ramamritham, and J. A. Stankovic. Implementing fault-tolerant services using the state-machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [8] K. Krithi, K. Ramamritham, and J. A. Stankovic. A local area network architecture for communication in distributed real-time systems. *The Journal of Real-Time Systems*, 3(2):115–147, May 1991.
- [9] L. Rodrigues and P. Verissimo. *A posteriori* agreement for fault-tolerant clock synchronization on broadcast networks. In *Twenty Second International Symposium on Fault-Tolerant Computing*, pages 527–536, Boston, Massachusetts, July 8-10 1992.
- [10] L. Rodrigues, P. Verissimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Thirteenth International Conference on Distributed Computing Systems*, pages 541–550, Pittsburgh, Pennsylvania, May 25-28 1993.
- [11] J. S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24(10):8–15, October 1986.