Processor Group Membership Protocols: Specification, Design and Implementation

Farnam Jahanian
Sameh Fakhouri

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Ragunathan Rajkumar

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents the specification, design and implementation of a set of protocols to solve the processor group membership problem *in distributed systems. These group membership protocols were developed as part of a toolkit for building distributed/parallel applications on a cluster of workstations. The group membership service forms the lowest layer in the toolkit, and it is the glue which unifies all other layers. The membership service supports three distinct protocols:* `weak`, `strong`, *and* `hybrid`. *These protocols differ significantly in the level of consistency and the number of messages exchanged in reaching agreement. The paper also describes the modular implementation of these protocols and the optimization techniques used to enhance their performance.*

1 Introduction

In a distributed environment, a collection of processors (or processes) can be grouped together to provide a service. A server group may be formed to provide high-availability by replicating a function on several nodes or to provide load balancing by distributing a resource on multiple nodes. In this paper, we focus upon the protocols for agreement among members on group membership in distributed systems.

This paper presents a suite of processor group membership protocols for asynchronous systems and describes their implementation. This suite consists of three protocols: weak, strong and hybrid membership. These protocols were implemented as part of a toolkit for developing distributed/parallel servers on a cluster of workstations. This toolkit is currently being used in several experimental projects in applications ranging from database servers to automated manufacturing and process control systems. The objective of this paper is to describe the motivation for the proposed processor membership protocols and to present a short overview of each protocol with some insight into our experience in building a prototype. A more detailed discussion on the formal specification of these protocols and their implementation appears in [10].

1.1 Motivation

A group membership protocol is an agreement protocol for achieving a *consistent* system-wide view of the operational processes in the presence of member departures, member joins and communication failures. The membership of a group can change when a member joins, a member departs, or a member is perceived to depart due to random communication delays. A member may also depart from a group due to a normal shutdown, such as scheduled maintenance, or due to a failure. Since it is impossible to detect correctly the crash of a processor in an asynchronous environment, in certain cases, a member departure is in fact an assumed failure by the other members. A join can occur when a new member is added to a system or when a failed member is restarted.

The *group membership problem* has been widely studied in recent years both for synchronous and asynchronous systems [1, 3, 4, 5, 6, 9, 11, 13, 14]. A number of experimental and commercial systems have implemented variations of the group membership service [2, 3, 7, 12]. Membership services are also used in managing replicated data and system availability in applications such as databases, distributed system management and monitoring, and process control. The degree of consistency among the views of the members has been a primary motivating factor for our work in proposing a suite of membership protocols.

The group membership service in our system supports three protocols: weak, strong, and hybrid membership. The proposed protocols differ significantly in the level of consistency and the overhead in reaching agreement when a membership change occurs. These protocols allow the architect of a system to balance the degree of consistency and the overhead of the group membership service based on the semantics required by different applications. The consistency required of a membership protocol is determined by the the semantics of the layer that uses the service.

Informally, the weak processor membership guarantees eventual agreement on the group membership when the system stabilizes after departures or joins.

The strong membership protocol ensures that the operational processors see the same sequence of group membership changes in the system. The hybrid membership is a compromise between the other two protocols. While it preserves a partial order on membership changes, it does not guarantee that the same sequence is observed by the operational members. However, it does ensure a unique group leader in the system.

We believe that different levels of consistency are required in different applications of group membership services. Most existing implementations of group membership protocols are variations of what we refer to as the strong membership: the changes in membership are seen in the same order by all members. A weaker notion of consistency is acceptable in many applications. For example, distributed system management and monitoring tools often require a much looser notion of agreement when a processor joins or leaves a group. The same may be true for a client's view of the servers in a parallel database machine connected to the clients via redundant communication networks.

The processor group membership service described in this paper is part of a toolkit of primitives for building distributed/parallel applications on a cluster of closely-coupled processors.[1] This toolkit consists of a layered set of services and has been used in the construction of highly-available servers for several applications including databases and automated process control.

1.2 Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we introduce our system model and its failure semantics. The notation used and some basic definitions are also presented. In Section 3, we present an overview of the three group membership protocols. In Section 4, we describe our implementation of these protocols, and the modularity and optimization techniques that we used to enhance the portability and performance of the protocols. Finally, we present our concluding remarks in Section 5.

2 System Model and Definitions

2.1 System Model & Failure Assumptions

In this paper, we consider an asynchronous *distributed system* consisting of a set of processes **P** running on processors connected by a communication network. The dependency between the local process states is caused exclusively by message communication. No assumption is made about the network topology or the protocol for implementing the communication service. However, it is assumed that each process in **P** can send messages directly to the other processes. The communication subsystem can suffer a *performance* or an *omission* failure. The message delay between the moment a message is sent by a source process and the moment the message is received by its target process is potentially unbounded. A message delay depends on many random factors including the computation and communication load of the system, message retransmission due to corruption or loss, and other O.S. events such as page faults and context switches.

We make no assumption about a global system clock or synchronized processor clocks in our computation model. However, we do assume that the relative rate of change (the drift rate) between two clocks is insignificant for intervals of a few seconds duration. This assumption can be stated informally as follows. When a processor p_i observes, say, 3 seconds in its local clock, another processor p_j does not see the passage of a 15 second interval its local clock. This does not imply that the processor clocks are approximately synchronized and see roughly the same absolute time. In fact, the deviation between processor clocks may grow without a known bound. Without the preceding assumption, the strong membership protocol would require an extra phase which would result in a 3-phase protocol instead of a 2-phase protocol, as described in Section 3. We also assume that stable storage is available to store unique group ids.

In this model, processes are assumed to have *crash* failure semantics. Since the detection of a process (or processor) crash is done via timeouts, it is impossible to determine with absolute certainty whether a process has crashed in an asynchronous system. For example, when a number of heartbeat messages from a process are lost due to system overload (*performance error*) or transmission error, the process may be seen to have failed by other processes. When a member is perceived to have crashed by the other members, the protocols ensure that the member exits and may rejoin the group in a consistent manner. Hence, a perceived crash of a group member is simulated as a crash failure in the system. The exact behavior of the group members varies from one protocol to another.

2.2 Preliminary Definitions

Before specifying the properties of the group membership protocols in the subsequent sections, a few preliminary definitions are presented. The protocols specify a processor id as a pair *(pid, instance)* where *pid* is a unique identifier for the processor and *instance* is the incarnation number for the processor[2]. When a collection of processors form a group, a unique id *gid* is used for identifying the group. As processors leave (or fail) or join a group, a new unique group id is assigned to capture the membership change.

Membership States and Transitions:
The *local membership state* of a processor p_i is defined by a pair of state variables $s_i : (gid, gmem)$

[1] We use the term *closely-coupled processors* to refer to a collection of (perhaps homogeneous) processors connected by a high-speed communication network that behave as a single server to the clients.

[2] In the implementation, we used the processor IP address as its pid.

such that $gid \in G \bigcup \emptyset$ and $gmem \in P^*$ where G is the set of all possible group identifiers and P^* is the power set of all possible processor ids. The symbol $\emptyset$ denotes a special null state.

The *global membership state* of a system is defined to be a vector $S: (s_1, s_2, s_3, ..., s_n)$ where s_i is the local membership state of processor p_i.

A *local membership state transition* of a processor p_i is an atomic assignment to the membership state of the processor denoted by an ordered pair of local membership states: $s_i \rightarrow s'_i$

A *global membership state transition* of a system is defined to be an ordered pair of global membership states $S \rightarrow S'$ such that $S = (s_1, s_2, ..., s_j, ..., s_n)$ and $S' = (s_1, s_2, ..., s'_j, ..., s_n)$ and $s_j \rightarrow s'_j$ is a local state transition of a processor p_j.

A *membership run* of a system is defined to be a sequence of global states $S_1, S_2, S_3, ...$ such that $\forall i \geq 1 S_i \rightarrow S_{i+1}$ is a global state transition.

Observe that properties of different membership protocols are specified as legal membership runs. First, we need to define several state predicates that are useful for specifying the properties of each membership protocol in the following section.

Definitions: Suppose s is a local membership state of a processor p in a global membership state S. The following predicates are defined on the state $s = (gid, gmem)$.

$joined_p(g)$	holds in state S if $gid = g$
$q \in members_p(g)$	holds true if $q \in gmem$
$q \in leader_p(g)$	holds true if q is the highest member id in $gmem$

Membership Precedence Relation:
Given a domain **G** of all possible group identifiers, we define a binary relation $\prec$ on the elements in G. For a membership run of a system, S_1 , S_2 , S_3 , ...

$g \prec g'$ iff $\exists S_i, S_j$ such that $S_i \models firstjoined_p(g) \wedge$t $S_j \models firstjoined_p(g') \wedge i < j \wedge g \neq \emptyset \wedge g' \neq \emptyset$

$S_i \models firstjoined_p(g)$ if $joined_p(g)$ holds in state S_i and $forall 1 \leq k \leq i$ $joined_p(g)$ does not hold in S_i.

We define the binary relation $\overset{*}{\prec}$ to be the transitive closure of $\prec$ relation. In the specification of the properties of the protocols, the temporal operators $\Box$ and $\Diamond$ are used to denote *always* and *eventually*, respectively.

3 Overview of Membership Protocols

The processor group membership problem is implemented by a set of daemon processes each running on a single processor, as shown in Figure 1. The failure of a daemon process is viewed as the failure of the processor. Informally, the objective is to provide agreement among group members (i.e. daemons) in the presence of processor joins, processor departures and communication failures. A client process may subscribe to the local membership daemon to be notified of a membership change in the system.

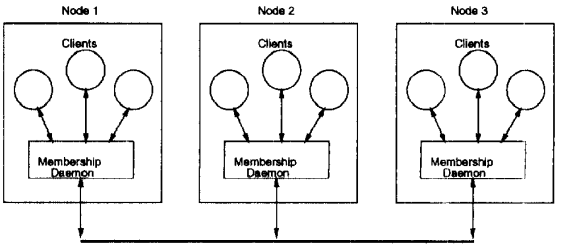

Figure 1: Membership Daemons and their Clients.

3.1 Common Features of All Three Membership Protocols

All three protocols namely the weak, strong and hybrid protocols share many common features, as discussed in greater detail in Section 4. Every processor gets assigned a unique identifier based on its network address and an instance number that changes each time the processor changes state relative to the membership system. Using these identifiers, a group leader is chosen, and a logical ring structure is created. Every member of a group has two neighbors in the ring, and it exchanges heartbeats only with those two neighbors. Failures are detected on the basis of missed heartbeats. Every group has a unique group identifier: this identifier is simply the tuple identifying the group leader and its instance. Also, each group has a member which will be the leader of the new group obtained by eliminating the leader from the old group. This member is referred to as the *crown prince*.

3.2 Safety Assertions

Certain safety assertions must be ensured by a group membership protocol. Two important assertions satisfied by all three protocols in this paper are:

SA1. If a processor p joins a group g, p is a member of the group:

$\Box[\ joined_p(g) \rightarrow p \in members_p(g)\ ]$

SA2. If two processors, p and q, join the same group g, both processors see the same members in the group:

$\Box\ [joined_p(g) \wedge members_p(g) = m \wedge$
$\Diamond [joined_q(g) \wedge members_q(g) = m'] \rightarrow m = m']$

3.3 Weak Group Membership Protocol

The weak group membership protocol simply guarantees that if there are no failures for some period of time, the views of all processors converge to a single consistent group in the system. Informally, the protocol ensures eventual agreement on the group membership when the system stabilizes after departures, joins or communication failures. As shown in Figure 2, it is possible that a membership change is not seen by a member, but eventually all operational members converge to the same view. The proposed weak membership protocol is a 1-phase protocol.

Properties of Weak Membership:

The binary relation $\prec$ on the set G of all group identifiers was defined in the preceding section. Informally, $g_i \prec g_j$ if a processor p joins the group g_i before joining g_j. The binary relation $\overset{*}{\prec}$ is defined to be the transitive closure of $\prec$ relation. The weak membership protocol ensures that the binary relation $\overset{*}{\prec}$ is a partial ordering relation.

W1. The relation $\overset{*}{\prec}$ is a partial ordering relation:

i) irreflexive: $g_i \not\overset{*}{\prec} g_i$

ii) transitive: $g_i \overset{*}{\prec} g_j \wedge g_j \overset{*}{\prec} g_k \rightarrow g_i \overset{*}{\prec} g_k$

iii) asymmetric: $g_i \overset{*}{\prec} g_j \rightarrow g_j \not\overset{*}{\prec} g_i$

This is a very weak guarantee, and it is relatively easy to achieve. The major complication is to ensure the asymmetric property in a 1-phase protocol. Specifically, we need to ensure that if a processor first joins g_i and then joins g_j, another processor does not join the two groups in the opposite order.

The weak membership protocol also ensures that the views of distinct processors eventually converge to the same membership state in the absence of failures. This liveness property guarantees that distinct group views, for example, when there are multiple concurrent failures and joins, eventually coalesce into a single group view. This property also ensures that after a network partition is repaired, multiple group views converge.

W2. If a processor p is a member of a group g_i and a processor q is a member of another group g_j, then unless there is a failure, there is eventually a state in which both processors are in a group g_k such that g_k succeeds g_i and g_j.

$\Box[joined_p(g_i) \wedge joined_q(g_j) \rightarrow$
$\Diamond[\exists g_k \; joined_p(g_k) \wedge joined_q(g_k) \wedge g_i \overset{*}{\prec} g_k \wedge g_j \overset{*}{\prec} g_k]$

Protocol Description:

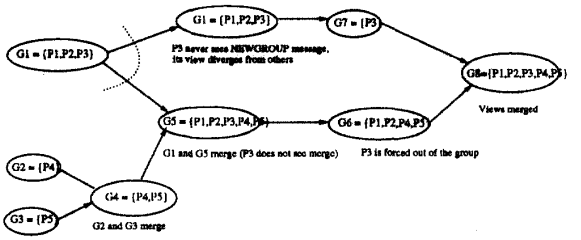

Figure 2: Partial ordering of group changes in weak membership.

Group members exchange heartbeats in two directions around a logical ring. When a new member (or members) request to join or a member failure is detected, a new group is formed. The leader forms a new group by simply sending out a **NEW_GROUP** message to all of the members. A processor that receives one of these messages can either join this group or not. Normally, it would join the group; however, if the message comes from a processor that was not a member of the old group, then it is necessary that this new leader should know which group each member is currently in. Specifically, this prevents interleaving of **NEW_GROUP** messages.

If a member fails to receive a new group message (because of message losses or long delays), then it will not exchange correct heartbeats with the members of the new group. The group will be re-formed without this member. However, it is possible for one member to have a different membership view from others temporarily. The member isolated with the inconsistent view will form a singleton group, and it will be allowed to join the others eventually.

Inconsistencies may occur when two groups merge, or when a member dies. All that happens when a member dies is that the member noticing this notifies the leader of this death. If the leader does not respond within a certain period of time, this member assumes that the group has failed and forms a new singleton group. If the member that has died is the group leader, the member noticing this notifies the crown prince. Once again, if a new group is not formed within a certain period of time, the member forms a new singleton group. In short, the tolerance for failure or message loss is small, and the standard method of dealing with presumed multiple failures is withdrawal from the group. The bootstrap procedure, merging divergent group views, and member death recovery are the three major functions of this protocol. This makes the protocol relatively straightforward.

Example: Figure 2 illustrates partial ordering of group changes in the weak membership protocol. Suppose processors P_1, P_2 and P_3 agree on the same group view G_1. Furthermore, suppose processors P_4 and P_5 also join together to form a group G_4 after being started concurrently. In the absence of a partition or a failure, the weak membership protocol guarantees that the two groups will merge into a single group G_5 consisting of all five processors. If processor P_3, for example, does not see the group view G_5 because of message loss, the

views of the rest of the processors will diverge from the view of processor P_3. One possible outcome is the formation of two concurrent groups G_6 and G_7 for a short time. Once P_3 can communicate with other processors, the groups will eventually merge to the same view G_8 consisting of all processors.

3.4 Strong Group Membership Protocol

Informally, the strong group membership protocol ensures that membership changes are seen in the same order by all members. The strong protocol differs from the other protocols mainly in that all group changes are acknowledged. In the weak protocol, the leader simply sends out the new group. As mentioned, the possibility arises that some of the recipients of the `NEW_GROUP` message will never get the message or the leader fails while forming a new group. If this happens, an inconsistent view of membership may exist among the operational members temporarily. In the strong protocol, instead of sending a `NEW_GROUP` message, the leader executes a 2-phase protocol. The leader sends a `PTC` (prepare to commit) message when a new group is being formed. A processor, upon receiving this message, if the message is from a valid leader, removes itself from its old group. At this point, the group of this processor is said to be in a `IN_TRANSITION` state, i.e. it is a member in transition from one group to another. The group id associated with the local membership state of the processor is defined to be $\emptyset$ when the processor is in `IN_TRANSITION`. This processor then sends an `ACK` message to the leader. The leader, after collecting either `ACK`s or `NAK`s from all the members, or when it has timed out waiting, determines what the membership of the new group will be. It then sends out a `COMMIT` message containing the group membership to all the members. The important aspects of this protocol are that the group changes are acknowledged, and that for some period of time, all the members that will be in a new group are in transition.

Properties of Strong Membership:

The strong membership protocol satisfies the safety properties **SA1** and **SA2** presented in Section 3.2. Furthermore, every membership run allowed by the strong membership protocol also satisfies the properties of the weak membership. Specifically, the strong membership ensures that the $\overset{\star}{\prec}$ precedence relation is a partial ordering relation (i.e., property **W1**). It also guarantees the liveness property that disjoint views of two processors eventually merge unless there is a failure (i.e., property **W2**). The strong membership protocol further restricts membership changes to ensure several other properties.

S1. In any global state, if a member p joins a group g locally, all other processors in g either joined g locally or are in a $\emptyset$ transition state:

$$\Box[firstjoined_p(g) \rightarrow \forall q \in members_p(g) \; joined_q(g) \vee joined_q(\emptyset)]$$

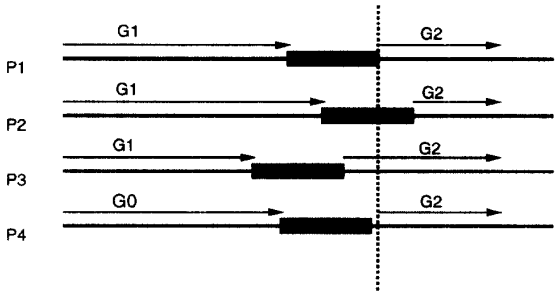

Figure 3: Example of Membership Changes in the Strong Membership Protocol.

The above property guarantees that while forming a new group g, there is a time interval during which every member of g has exited its former group and it is waiting to join g. The local membership state of a process is defined to be $\emptyset$ when the processor is in transition. The strong membership protocol forces a potential member of a new group into the $\emptyset$ state during the first phase of the protocol indicating that the processor is in transition from one group to another. The member then joins the new group by changing its local state (or local view) in the second phase. We will illustrate the above property by an example.

Example: Suppose three processors P_1, P_2 and P_3 agree on a group view g_1 consisting of the three processors. Figure 3 illustrates the membership change as seen by each processor when the processor P_4 joins to form a new group g_2 consisting of all four processors. The dark rectangle illustrate the `IN_TRANSITION` state denoting the transition interval from the time that a processor acknowledges to leave its existing group to the time that its local state reflects the membership change to g_2, i.e., a *commit* message from the group leader arrives. Observe that different processors may see their local state change at distinct times. The dotted vertical line illustrates how property **S1** is observed by processor P_1. When processor P_1 sees the new group g_2, the local membership state of each other processor either has already been changed to g_2 or it is in transition. Hence, when P_1 changes its local membership state to g_2, processor P_2 is in transition while processors P_3 and P_4 have already seen the membership change.

Handling Partitions:

We introduce the notion of *active* and *inactive* groups to deal with partitions in the system. When a processor first comes up, it joins a singleton group consisting of itself. That group is initially designated as an *inactive* group. If there are no other processors around, the inactive group will eventually become *active*. If there already exists an active group, the singleton inactive group will merge with the active group and thus inheriting its active status[3]. Ensuring agreement

[3]The implementation of this membership service allows specifying other policies, such as the majority rule or the existence of a particular set of processors for deciding to go

on group membership is complicated because of performance failures due to the loss of messages, transient processor overload, hardware failures such as the loss of a communication adapter or a link failure. When a member suffers a performance failure (for example, by failing to send its heartbeat), other group members may *presume* that the processor suffering from a performance failure has died. The membership views may be inconsistent if multiple (overlapping) active groups can exist concurrently.

The objective is, of course, to ensure the existence of a single *active* processor group in the system. It can be shown that this objective is impossible to achieve in an asynchronous distributed system, since the problem can be formulated as a variation of the distributed consensus problem [8]. However, the strong membership protocol ensures that concurrent (active) groups, in the presence of transient partitions or performance failures, are disjoint. Furthermore, the relation $\overset{a}{\prec}$, defined as the projection of the precedence relation on the active groups, imposes a total order on the active groups in a membership run in the absence of performance failures.

S2. In the presence of performance failures or network partitions, members of concurrent active groups are disjoint.

S3. *Total Order on Active Groups:* In the absence of performance failures or partitions, the binary relation $\overset{a}{\prec}$ totally orders the active groups in G.

Protocol Description:

As with the case of the weak membership protocol, group members exchange heartbeats in two directions around a logical ring. When a new member (or members) requests to join or a member failure is detected, a new group is formed by reaching agreement on the members of the new group. When a member dies, some neighbor notices the failure to send heartbeats, and it notifies the leader that its neighbor has died. Once again, it sets a timer to control how long it will wait for notification of a new group. In particular, the current implementation will notify the leader of the death several times before it finally abandons the group and forms its own singleton group. If the leader successfully forms a new group after the death of some member, it will perform the same steps as if two groups were being merged; the `PTC` → `ACK` → `COMMIT` sequence will occur normally. In the event that the leader has died, the neighbor notifies the crown prince as in the weak protocol. The crown prince then executes a 2-phase protocol forming a new group.

A possible inconsistency that might occur is that if the leader has not actually died, but has simply lost communications with one node, e.g., the crown prince. The leader and the crown prince may assume that the other one is dead and each may attempt to form a new group. This failure mode, unless avoided, would result in inconsistent views.

In an asynchronous system, it is provably not possible to avoid all cases where the leader may think some member has died, and that member may think the leader died. All that can be done in this case is to make this sort of partition not cause group inconsistencies. For example, suppose the members of a group G_1 agree on its membership: $G_1 = (P_{23}, P_{19}, P_{14}, P_2, P_1)$. Because of loss of heartbeat messages, P_{19} and P_{23} (the leader) mark each other as dead. While P_{19} is taking over, we might have P_{23} also forming a new group. The end result might be two groups: $G_2 = (P_{23}, P_{14}, P_1)$ and $G'_2 = (P_{19}, P_2, P_1)$ where P_1 is a member of both groups. This can happen if the messages to P_1 interleave in just the wrong way. The result is that G'_2 will fail; these situations can result in both groups continuously failing and being restarted indefinitely. The worst side effect is that both groups now have a majority! This sort of oscillation between two groups must obviously be avoided.

One way to avoid such oscillation is with the unique group identifiers. In the above case, P_1 was in one of the two groups first, and it then moved to the other group. If every group transition is made dependent on the new leader correctly specifying what group the members *were* in, then this problem would be avoided. In the above example, both P_{19} and P_{23} would try to cause P_1 to make a transition from group G_1 to some new group. After one of them succeeds, the other will fail since P_1 would no longer be in G_1 but would be in some new group. This simple, low overhead solution avoids this problem altogether. The code to do this is implicit in the algorithm pseudo-code for all three protocols; a message that arrives with the wrong group is handled as an error condition, and an error message is returned to the sender.

The above scenario is not unique to the death of a group leader. It may happen whenever one member presumes the death of another member due to the random loss of heartbeat messages, transient network partitions or a processor overload. The strong group membership protocol in this paper ensures the formation of *non-overlapping* groups in those cases. This behavior is ensured by a 2-phase protocol (instead of requiring a 3-phase protocol) because of the assumption on the clock drifts, as mentioned in Section 2.1.

The proposed protocols for strong group membership is presented below. The protocol consists of a sequence of initialization steps followed by a loop that continuously processes incoming messages and handles various timer interrupts. In the event of serious errors, when it is necessary to re-initialize, all that is required is to break out of the inner loop and start at the top.

```
START:
  initialize to singleton group;
LOOP:
  while(true) {
```

from an inactive group to an active group.

```
  wait(message or timer);
  switch(message.kind) {
    case PROCLAIM:
    case JOIN:
    ...
    case STATCHANGE:
  }
  if (heartbeat_expect_timer.expire) ...
  ...
  if (proclaim_timer.expire) ...
}
```

When a processor goes through initialization, the processor id and its latest instance number are retrieved from stable storage. The instance number is stored on stable storage after it is incremented by one. A singleton group is formed with the processor as its leader. The variables G and L denote the local view of the group membership and its leader. The pseudo-code for the initialization steps is shown below:

```
START:
  read(my_id);/* read my id (pid,instace) */
  set my_id; /* set new id:  inc instance# */
  /* set my group, its leader and status */
  G ← ( my_id );
  L ← leader(G);
  status ← INACTIVE;
  /* set timer for becoming ACTIVE */
  set status_timer;
  /* send to core mems not in G */
  send(PROCLAIM,core - G);
  /* set timer for proclaim sends */
  set proclaim_timer;
```

As discussed earlier, the main loop handles each incoming message based on the *message kind*: PROCLAIM, JOIN, PTC, ACK, NAK, COMMIT, HEARTBEAT, and STATCHANGE. The following code segments illustrate how each message kind is handled.

```
case PROCLAIM:
if (leaderp) { /* am I the leader?  */
  /* is sender higher priority */
  if (message.fromid ≥ myid) {
    /* join his group */
    send(JOIN,G,message.fromid);
    G ← NULL;
    waiting ← true;
    set waiting_timer;
  } else /* he should join me */
    send(PROCLAIM,message.fromid);
} else /* forward proclaim to leader */
  forward(message,L);

case JOIN: ...
case PTC:
/* ack the prepare to commit */
send(ACK,message.fromid);
unset all timers /* stop all timers */
committing ← true;
G ← NULL
set ptc_timer;

case ACK: ...
case NAK: ...
case COMMIT:
if (committing && L != my_id) {
  /* set the group list, leader, and status */
  G ← message.group;
  L ← leader(G);
  status ← message.status;
  /* set last active group */
  last_active_group ← message.last_active_group;
  unset all timers except heartbeats;
  committing ← false;
}

case HEARTBEAT:
/* reset heartbeat timer */
reset heartbeat_expect_timer;
/* we haven't missed one */
heartbeat_missed ← 0;

case DEATH_IN_GROUP: ...
case STATCHANGE: ...
```

The main loop also handles various timers that may expire. The timers in the strong protocol are: status_timer, heartbeat_expect_timer, heartbeat_send_timer, proclaim_timer, waiting_timer, dying_timer, and joining_timer. The following code segment illustrates the response to the above timers.

```
if (heartbeat_expect_timer.expire) {
  heartbeat_missed++; /* missed one more */
  if (heartbeat_missed ≥ LIMIT) {
    /* we have missed too many */
    if (neighbor == L) /* leader died?  */
      send(DEATH_IN_GROUP,crownp);
    else {
      send(DEATH_IN_GROUP,L);
      set dying_timer;
    }
  } else /* try waiting again */
    set heartbeat_expect_timer;
}

if (heartbeat_send_timer.expire) {
  send(HEARTBEAT,neighbor);
  set heartbeat_send_timer;
}

if (joining_timer.expire) ...
if (status_timer.expire) ...
if (waiting_timer.expire) ...

if (ptc_timer.expire) {
  /* never got a commit */
  goto START;

}

if (proclaim_timer.expire) {
  /* send to core mems not in G */
  send(PROCLAIM,core - G);
```

```
}

if (dying_timer.expire) {
  /* we never heard from leader about death */
  goto START;
}
```

3.5 Hybrid Protocol Description

The hybrid protocol is intended to provide a level of consistency between that of the strong and weak protocols. The hybrid membership protocol satisfies the safety properties **SA1** and **SA2**. Furthermore, the protocol preserves the partial ordering of group membership changes (property **W1**). Furthermore, it guarantees that there will be a single leader for any group. The way this protocol insures this is by running a multiphase commit whenever the leader of a group changes. Leader changes occur when two groups merge, or when the leader of a group dies. When two groups running the hybrid protocol merge, one of the two leaders becomes the new group leader. The members of the (inferior) group with a new leader are forced to run a multiphase commit. In other words, those members of the new group that are seeing a leader transition will do a `PTC` → `ACK` → `COMMIT` procedure. This ensures that the members of the group that is being done away with will have acknowledged the new leader. The members of the group whose leader didn't change will be informed similar to the 1-phase protocol. This multiphase protocol is not necessary when a non-leader member dies, since the death of the member does not cause any leader changes. The sequence of steps necessary when a member dies under the hybrid protocol is identical to what happens under the weak protocol. Under the hybrid protocol, the sequence of steps performed when the leader dies is the same as under the strong protocol. Further details on the hybrid protocol are omitted in this paper.

4 Implementation of the Membership Service

The weak, strong and hybrid membership protocols share many common features. This section focuses on several common components in the implementation of the three membership protocols. These include the assignment of processor and group identifiers, the way groups are structured, the way failures are detected, the way bootstrapping is carried out, the kinds of messages exchanged (to a large extent), and the way logical connections are maintained and mapped to real connections. The architecture of the processor membership service is illustrated in Figure 4. Each of these modules is described in greater detail below.

4.1 Identifiers and Group Structure

All three protocols understand two different types of identifiers: processor identifiers and group identifiers. Processor identifiers uniquely identify all processors in the system, and group identifiers uniquely identify all possible group views in the system. In our implementation, both of these identifiers have the same form. An identifier is the 2-tuple *(processor address, instance number)*, where the address is assumed to uniquely identify all processors in the system, and the instance number distinguishes between different "incarnations" of a processor. In the TCP/IP implementation, the IP address of the processor is chosen as its unique address. For example, a processor id might be *(0x80209008,0x8)*. The first part of this id indicates that this is the machine *foo.watson.ibm.com* with the IP address 128.32.144.8. The second part of this address indicates that this is the eighth unique occurrence of this processor membership daemon. The instance number is incremented every time the processor is the leader of a new group. As will be seen, this includes such occurrences as restarts and groups dissolving. A group identifier is just the processor identifier of the processor that is the leader of this group. Since every new group causes the group leader to have a new unique id, every group has a unique id.

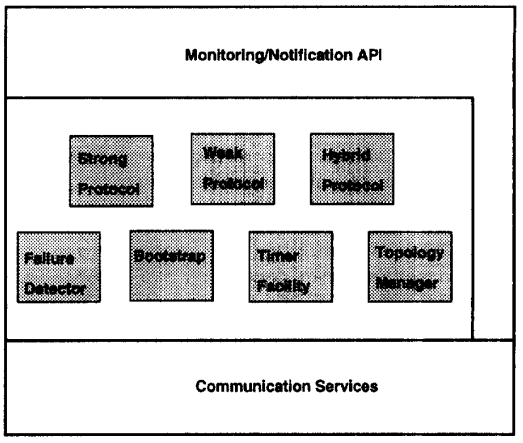

Figure 4: The Software Layering of the Membership Service

A processor group consists of some number of members, i.e. processors. One of these members is designated as the group leader. Currently, the member of a group with the highest processor identifier, i.e., the largest IP address, is designated as the group leader. While the criteria used to determine the leader is not important, it is critical that the leader be unique. It is also crucial that every processor can determine the leader of a group simply from the list of group members. Both of these properties are relatively easy to insure given the form of the identifiers used. In addition to a group leader, every group with more than one member has a member designated as the *crown prince*. This member will become the leader of the new group obtained by eliminating the leader from the current group. In other words, consider the group $G_1 = \{P_1, P_2, P_3\}$ with P_3 as its leader. If P_3 were removed from the group resulting in the group $\{P_1, P_2\}$, then P_2 would be the leader of this group. Hence, P_2 is referred to as the crown prince of G_1.

4.2 Heartbeats and Failure Detection

All three protocols use heartbeat messages to determine which processors are up and which are not. The heartbeat behavior is configurable to control message loss tolerance and the time necessary to detect a failure. The heartbeat behavior is defined by two timers and a counter. The timers, `heartbeat_expect` and `heartbeat_send` control the frequency with which heartbeats are expected and sent respectively. The relationship `heartbeat_expect` $\geq$ `heartbeat_send` should be true. Otherwise heartbeats will be marked as missed constantly and many false failures will be reported. The counter, `heartbeat_missed`, counts how many heartbeats are missed. If this exceeds some constant, the processor will report the neighbor that has failed to send the heartbeats as potentially dead. When the `send timer` expires, the processor sends a heartbeat to its neighbor. In a correctly functioning system, the `expect timer` should never expire as it is reset every time a heartbeat from the neighbor is received. The counter is also reset every time a heartbeat from the neighbor is received. If the `expect timer` does expire, the count is incremented, and the timer is reset. If the count is too large, the failure is reported. Obviously, as the period of the `expect timer` is made larger, the time to detect a failure increases. However, a balance between number of messages and failure detection time must be reached. Similarly, as the number of missed heartbeats allowed is increased, the time to detect a failure increases, but the likelihood of reporting a false failure due to a random message loss decreases. In our experiments, we have used `send` and `expect timers` ranging from 250 milliseconds to several seconds. For example, in a closely-coupled cluster for an automated manufacturing and process control application, the `expect timer` was set to 250*ms* with a missed heartbeat count of 2. This profile requires frequent heartbeat between neighbors with the period of 125ms. In an 8-way parallel database machine prototype connected by a 16Mb network, the strong processor membership protocol is currently used with a `send timer` of 1 second and an *expect timer* of 2 seconds. The time to detect a failure could be as much as 4 seconds assuming a missed heartbeat count of 2. But the number of heartbeat messages is very low in this case, and the load on the processors and network turns out to be less than 2 percent. Furthermore, the number of false failures reported is practically zero in this system.

The logical connections between members in a group control the structure of the group to a large extent. In general, messages such as heartbeats are sent not to the entire group, but only to some subset of the group. The logical connection scheme used determines which subset of the group such a message would be sent to. For modularity and ease of experimenting with different topologies, the logical connections between members is maintained by a topology/configuration component. One of the simpler connection schemes is a logical ring. A logical ring maps the members of a group on to some arbitrary ring structure. The topology/configuration component implemented in our system maintained a doubly-connected logical ring among group members. Each members is required to send a heartbeat message to both neighbors on the left and the right. The redundant heartbeats were used at run-time to reduce the probability of false failure detection.

4.3 Bootstrap

When a number of processors first come up, there is a basic problem that involves determining who else, among all of the processors that might potentially be in the system, is up. This information is needed to allow a processor that comes up to merge with an existing group of processors. There are two customary ways of solving this problem: (a) send a broadcast message to all possible nodes in the network, or (b) maintain a list of all processors in the system. A broadcast-based scheme is feasible if there is hardware support for sending a broadcast message. Maintaining a replicated list of all proocessors on each node is an agreement problem that in turn requires the knowledge of who is up and down. To break this loop, some systems have simply said that this list should be maintained by an outside agent (such as a system administrator) or cross-mounted.

Our implementation does not assume hardware support for multicast and it attempts to minimize the need for maintaining identical copies of the processor list on all nodes. Our scheme is based on the observation that in most closely-coupled systems, a *core* subset of processors must be up before the system is operational. For example, in a parallel database server, the log servers must be up before the DB server is available. Similar statement holds for the name servers in a cluster of workstations. If all processors in the system have a list of the core members, and if at least one member in the intersection of these lists is up, then a processor in a singleton group can find out about others by contacting the core members first. These lists of core members do not have any particular consistency requirements, nor do they require atomic updates. In the worst case, the core sets of processors might contain the name of every other processor in the system, but in the case we optimize for, there is a small set of processors such that the system is not usable without at least one of these processors being up. Using this scheme, we have been able to seamlessly add new processors to a system without having to change the list of core processors. This notion of a core of processors reduces the number of messages required compared to broadcast-based schemes, and it significantly reduces the amount of human intervention required.

The weak, hybrid and strong membership protocols share a simple bootstrap procedure. A processor simply forms a group with itself as the only member when it comes up. The *instance number* associated with its processor id is retrieved from stable storage, is incremented by 1, and then it is written back to stable storage before the singleton group is formed. Recall that the new processor id is also the group id for the singleton group. This singleton group is the base for all subsequent groups: any time a processor encounters an error condition that cannot be dealt with in the context of the current group, it can simply reinitialize itself by forming a singleton group. The leader of a group,

whether singleton or not, sends a proclaim message to the core members. As was described above, so long as one processor in the intersection of the core sets is up, every group will eventually merge to form a single group. The frequency of the proclaim messages is the key parameter that determines how quickly the views of concurrent groups merge together.

4.4 Monitoring and Notification Facility

Client processes can subscribe to the processor membership service by using a monitoring and notification facility. As shown in 4, an application programming interface (API) provides a simple uniform interface for a client process to subscribe to the membership service. A registered subscriber will be notified by the membership layer whenever there is any change in processor membership. Two types of notification are available: *synchronous notification* and *asynchronous notification*. Under synchronous notification, membership changes will be sent to a communication port which is polled by a subscriber. Under asynchronous notification, a callback function in the subscriber's address space is invoked whenever a membership change occurs. The callback function is specified by the subscriber when it registers itself for asynchronous notification. A set of calls for querying the membership state of a system are also provided as part of the monitoring and notification API.

4.5 Modularity

The modular design and implementation of the membership daemon has proven to be very versatile and flexible. The total amount of code for implementing the membership service including the three protocols and the surrounding mechanisms was about 15K lines of C code. The modularity of its design has allowed us to experiment with new membership protocols with relative ease and it has facilitated porting the membership service to several operating systems with support for different communication protocols. To date, the membership daemon has been ported to variations of the UNIX operating system including AIX v.3.2 and BSD-compatible versions as well as OS/2 v2.0. [4]

The membership daemon was originally implemented with the weak protocol. The strong and hybrid protocols were added to the membership service a few months apart. Adding a protocol consists of defining the messages used by the protocol and handlers for each of the messages is written. Then, the protocols are designed as an infinite loop that sends heartbeats, checks incoming messages and calls appropriate message handling routines. This experience has given us considerable insight on how easy it is to add a new protocol to the membership service. Adding a new protocol was relatively a small effort that involved less than 700 lines of new code to be written.

[4] UNIX is a registered trademark of AT&T Bell Laboratories. AIX and OS/2 are trademarks of IBM Corporation.

5 Concluding Remarks

We have presented the specification, design and implementation of a suite of three group membership protocols with varying degrees of consistency and cost in reaching agreement in a distributed system. These protocols have been implemented on AIX v3.2 and on OS/2 2.0, but they can also run on any BSD-compatible system. The AIX implementation uses UDP datagram services while the OS/2 implementation uses NetBios transport layer. We are also currently experimenting with the use of these protocols on very fast point-to-point networks. We have addressed several implementation issues and optimization enhancements. These include handling of partitions and adding a new membership protocol. We also present a new approach to the problem of simultaneous startup of multiple processors as it relates to group membership. Finally, we discussed protocol optimizations that reduce the number of messages exchanged or decrease the latency of reaching agreement.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *6th Int. Workshop on Distributed Algorithms*, pages 292–312. Springer-Verlag, November 1992.

[2] A. Bhide, E. N. Elnozah, and S. P. Morgan. A highly available network server. *USENIX Summer Proceedings*, pages 199–205, June 1991.

[3] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

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

[5] B. A. Coan and G. Thomas. Agreeing on a leader in real-time. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 166–172, December 1990.

[6] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 6(4), 1991.

[7] F. Cristian and R. Dancey. Fault-tolerance in the advanced automation system. Technical Report RJ7424, IBM Almaden Research Center, April 1990.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[9] F. Jahanian and W. L. Moran Jr. Strong, weak and hybrid group membership. In *Workshop on Replicated Data Management II*, pages 34–38, Nov 1992.

[10] F. Jahanian, R. Rajkumar, and S. Fakhouri. Processor group membership protocols: Specification, design and implementation. Technical report, IBM T.J. Watson Research Center, August 1993.

[11] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a sync distributed real-time system. In *in Dependable Computing for Critical Applications, A. Avizienis and J.C. Laprie, ed.*, pages 411–429. Springer-Verlag, 1991.

[12] N. P. Kronenberg, H. M. Levy, W. D. Strecker, and R. J. Merewood. The vaxcluster concept: An overview of a distributed system. *Digital Technical Journal*, September 1987.

[13] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In *Proceedings of International Working Conference on Dependable Computing for Critical Applications*, February 1991.

[14] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 1991.