2

NPS-EC-92-009

NAVAL POSTGRADUATE SCHOOL

Monterey, California

GROUP MEMBERSHIP IN ASYNCHRONOUS DISTRIBUTED ENVIRONMENTS USING LOGICALLY ORDERED VIEWS

Shridhar B. Shukla
Devalla Raghuram

16 September 1992

422748 27pp

Approved for Public Release: Distribution Unlimited

Prepared for: Naval Postgraduate School
Monterey, California

This report was prepared for and funded by the Naval Postgraduate School.

This report was prepared by:

[illegible]

SHRIDHAR B. SHUKLA
Assistant Professor of
Electrical and Computer Engineering

[illegible]

for DEVALLA RAGHURAM

Reviewed by:

[illegible]

MICHAEL A. MORGAN
Chairman,
Department of Electrical
and Computer Engineering

Released by:

[illegible]

PAUL J. MARTO
Dean of Research

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS-EC-92-009		5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Dept. of Elect. & Comp. Eng. Naval Postgraduate School	6b. OFFICE SYMBOL *(if applicable)* EC/Sh	7a. NAME OF MONITORING ORGANIZATION NPS
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5004		7b. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NPS	8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

Group Membership In Asynchronous Distributed Environments Using Logically Ordered Views

12. PERSONAL AUTHOR(S)

Shridhar B. Shukla and Devalla Raghuram

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical Report	FROM 10/1/91 TO 9/30/92.	16 September 1992	22

16. SUPPLEMENTARY NOTATION The views expressed in this report are those of the author and do not reflect the al policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB--GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

Agreement, Asynchronous, Commit, Distributed, Failure, Group Membership, Logical Ring, Reliable Multicast, Token

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

A group membership protocol ensures agreement and consistent commit actions among group members to maintain a sequence of identical group views in spite of continuous changes, either voluntary or otherwise, in processors' membership status. In asynchronous distributed environments, such consistency among group views must be guaranteed using messages over a network which does not bound message delivery times. Assuming a network that provides a reliable, FIFO channel between any pair of processors, one approach to designing such a protocol is to centralize the responsibility to detect changes, ensure agreement, and commit them consistently in a single manager process. This approach is complicated by the fact that a protocol to elect a new manager with a consistent membership proposal must be executed when the manager itself fails. In this report, we present a membership protocol based on ordering of group members in a logical ring that eliminates the need for such centralized responsibility. Agreement and commit actions are token-based and the protocol ensures that no tokens are lost or duplicated due to changes in membership. The cost of committing a change is 2*n* point-to-point messages over FIFO channels where *n* is the group size. The protocol correctness has been proven formally.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Shridhar B. Shukla	22b. TELEPHONE *(Include Area Code)* (408) 646-2764	22c. OFFICE SYMBOL EC/Sh

Group Membership In Asynchronous Distributed Environments Using Logically Ordered Views[1]

by

Shridhar Shukla[2] and **Devalla Raghuram**[3]
Code EC/Sh. Dept. of Elect. & Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5004
Tel: (408) 646-2764 Fax: (408) 646-2760
E-mail: shukla@ece.nps.navy.mil

[1] This research has been supported in part by the Naval Postgraduate School, Monterey, CA 93943

[2] Responsible for all communication

[3] Currently on deputation from the Defense Research Development Organization, India, to pursue an MS at the Naval Postgraduate School under the IMET program

Abstract

A group membership protocol ensures agreement and consistent commit actions among group members to maintain a sequence of identical group views in spite of continuous changes, either voluntary or otherwise, in processors' membership status. In asynchronous distributed environments, such consistency among group views must be guaranteed using messages over a network which does not bound message delivery times. Assuming a network that provides a reliable, FIFO channel between any pair of processors, one approach to designing such a protocol is to centralize the responsibility to detect changes, ensure agreement, and commit them consistently in a single manager process. This approach is complicated by the fact that a protocol to elect a new manager with a consistent membership proposal must be executed when the manager itself fails. In this report, we present a membership protocol based on ordering of group members in a logical ring that eliminates the need for such centralized responsibility. Agreement and commit actions are token-based and the protocol ensures that no tokens are lost or duplicated due to changes in membership. The cost of committing a change is $2n$ point-to-point messages over FIFO channels where n is the group size. The protocol correctness has been proven formally.

1 Introduction

Consistent views of the membership of a group of entities that cooperate to perform a task is basic to construction of distributed applications using the process group approach [BSS91]. The group of entities may correspond to a set of processes that must behave consistently to provide a service or a set of processors that must determine their function based on which other processors are operational. Changes to the membership occur when members *fail* or *leave* the group and when they *recover* or *join* the group. Some form of consensus on group membership is necessary, for without it, a server that respects its specification may nonetheless behave inconsistently with respect to another server since they see different group members. The group membership problem refers to achieving such consensus. Its solution refers to a group membership protocol (GMP). Absence of shared memory in a distributed system requires a GMP to rely on message passing alone.

Typically, availability of a GMP supports construction of reliable communication primitives which in turn simplify construction of distributed applications. For example, guarantees about multicast communication in the presence of failures require an underlying GMP [B$^+$90, CM84]. Aside from the basic requirements of *safety* and *liveness*, a GMP can be evaluated in terms of how well it supports the required communication primitives. Prompt response to membership changes and ability to support changes continuously (*i.e.*, without stalling the application) are two of the desirable performance features of a GMP.

The design and complexity of a GMP depend critically on whether it operates in a *synchronous* or *asynchronous* distributed system. In the former, a GMP exploits tight synchronization among the clocks of the interacting processes and/or known upperbounds on message delivery times. It is possible for all application messages to wait till changes to membership are complete and for all membership changes to wait till all pending messages are sent. Examples of such GMPs are [Cri88, EdL90, KGR89].

In asynchronous systems, there is no relationship among clocks of the interacting processes and message delivery times are unbounded. Therefore, crashes are indistinguishable from communication delays or slow members. It is only possible to *perceive* failures. It is necessary that members perceived to have failed be removed from the group since it is impossible to reach consensus on a failure [FLP85]. In this report, we deal with GMPs for asynchronous systems only. The basic function of a GMP in an asynchronous system is to ensure that all operational members commit perceived changes to their local views consistently. The consistent commit entails agreement about the change perceived.

Several GMPs have been proposed for asynchronous systems. In [Bru85], failure/recovery detection and notification are achieved using successive message rounds. Maintaining consistent views is the responsibility of higher level software. The number of messages required scales nonlinearly with the number of members and the recovery protocol requires *a priori* knowledge of the potential members. Several GMPs are proposed in [LSA91] based on total ordering of messages. Such ordering has a high overhead cost and assumes a fault-tolerant, reliable broadcast communication protocol. In [CM84], reliable broadcasts are supported by rotating a membership list (token-list) among operational members. When a member holding the token list fails, a reformation phase is entered which guarantees that a single new token-list is generated and committed to by all members. During this phase, normal message traffic is suspended and handling of changes needs an extension to the protocol.

In [BJ87], a two-phase site-view management protocol is proposed to support higher level fault-tolerant communication primitives. Its drawback of blocking during continuous failures and recoveries is removed in the formal solution proposed in [RB91]. Assuming a completely connected network of reliable FIFO channels and *fail-stop* behavior of member processes, this GMP uses a two-phase algorithm for the basic membership update and a three-phase algorithm when the reconfiguration manager itself fails. Election of a new manager with a consistent membership proposal must avoid *invisible commits*.

In this report, we describe a GMP for asynchronous systems to support reliable communication primitives required for *virtually synchronous* process group approach of [BSS91]. All application level communication between members of a group is assumed to carry a view number. It is required that each increment of the view number be associated with successive views that differ in only one member. Using a fully connected network of reliable FIFO channels, the proposed GMP guarantees that a given view number is associated with the same membership at any operational member.

The proposed GMP eliminates the need for centralizing the responsibility of ensuring consistency of view changes as in [RB91] by maintaining the group view ordered as a logical ring at each member. Each member perceives the departure of a neighboring member and joining members enter on one side of a virtual marker whose position is maintained by all the members. Agreement and commit actions are achieved using tokens circulated along the logical ring. The protocol is able to regenerate lost tokens and ignore duplicate ones generated during its operation.

This report is organized as follows. In section 2, the terminology used in the description of the protocol is established and our assumptions are listed. In section 3, the algorithms used in this GMP are described. In section 4, the correctness proof is presented. The report ends with concluding remarks in section 5.

2 GMP Overview

2.1 Assumptions

The proposed GMP makes the following assumptions. A reliable FIFO communication channel between any two members that are operational is assumed. In other words, it is assumed that the network is never partitioned. All failures are assumed to be *crash* or *fail-stop* [Cri88]. This implies that a message sent will not be delivered only because of the receiver's failure. However, it may be arbitrarily delayed. Continuous changes to the membership are allowed; however, the changes are committed one at a time. A member gets added when a *join* request is processed and gets deleted when a *departure* is perceived. A group name is assumed to be public to those processes that may wish to join the group. A mechanism, whereby a process wishing to join a group can locate a site already running a member of the group it wants to join, is assumed to be available.

2.2 Overview

The proposed GMP guarantees that view changes and their sequence at each operational member are identical. Using a view number in all group-related communication guarantees

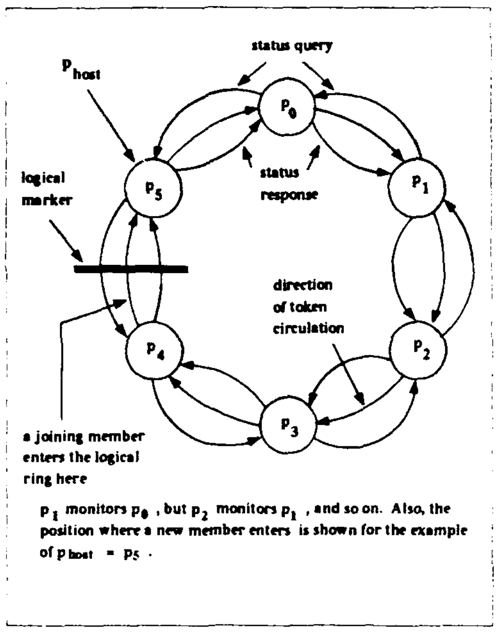
status query
p host
logical marker
P0
P1
P2
P3
P4
P5
status response
direction of token circulation
a joining member enters the logical ring here
p1 monitors p0, but p2 monitors p1, and so on. Also, the position where a new member enters is shown for the example of p host = p5.

Figure 1: A Logical ring

that reliable communication primitives can be built. The principle feature of this GMP is that there is no central element either to detect a change in membership status or to guarantee consistency of a commit action on the view of group membership. Both are achieved in a distributed manner using a logical ring which is simply a conceptual circular ordering of the members.

A logical ring has no relation with the physical locations of the members. Given such a ring and a direction of traversing it (arbitrarily, *clockwise* is selected), each member periodically queries its counter-clockwise neighbor for its status. The neighbor then responds with a status message when it receives this query. It, in its turn, sends a status query to its counter-clockwise neighbor. Thus, every member monitors one other member and is itself monitored by a third member. For example, if there are 6 members p_0 to p_5, a logical ring can be configured in which p_0 is an counter-clockwise neighbor of p_1 and clockwise neighbor of p_5, p_1 is an counter-clockwise neighbor of p_2 and clockwise neighbor of p_0, and so on. p_1 sends a status query to p_0 and p_0 responds with a status message to p_1. The status message from p_0 is monitored by p_1. This is illustrated in Fig. 1.

Initially, the ring configuration is known to all the members. As members change status, the ring configuration changes. The MP treats the cases of a member leaving the group in the same manner as a member joining the group. [1] The protocol maintains appropriate information at operational members to determine whom each member must monitor. When a member departs voluntarily, it simply stops responding to the status query from its monitor. If a failure occurs, it is unable to respond to its monitor. In either case, if a monitor does not receive a status message within a certain time interval after sending a query, the monitored member is perceived to have left the group. A sequence of actions to ensure that all the operational members consistently commit to this change is then invoked. When a member recovers or wishes to join anew, it sends a *join* request to the first group member it can locate. This member registers the request and invokes a sequence of actions, similar to that of departure processing, to ensure that consistent integration of the incoming member takes place.

2.2.1 Processing of Individual Changes

There are two phases in the protocol to process a join or a departure, *viz.*, the *agreement* phase and the *commit* phase. These phases are token-based and guarantee that each token is processed exactly once by each member and is never lost. Processing of individual view changes is described below. More detailed description of the actions taken in each phase is given in the next section.

Departure Processing:
Once a member perceives the departure of its monitored member because it does not receive a status message in response to its query for a predetermined time interval, it initiates the agreement phase by sending an agreement token to its clockwise neighbor. It also starts monitoring the counter-clockwise neighbor of the member perceived to have departed. The agreement token is passed around the ring in the clockwise direction by each member passing it on to its clockwise neighbor. When this token circulates back to the agreement initiator, it has gone completely around the ring once and all the operational members have information indicating that the group has reached an agreement on the departure perceived.

[1] **Failures amount to a member leaving involuntarily and recoveries amount to a member joining as a new one.**

The agreement initiator then starts the commit phase by generating a commit token which is circulated around the ring in the same manner as in the agreement phase. All the members receiving this token commit the change by removing the departed member from their group view and updating the view number.

Join Processing:
The protocol maintains a *logical marker* in the ring as the position between some pair of adjacent operational members at initialization. The clockwise member of this pair is designated as the host of the logical ring and is known to all members initially. As shown in Fig. 1, a new member always enters the group as the counter-clockwise neighbor of the host who has the responsibility of carrying out the agreement and commit phases for the new member. A member that receives a join request from a potential member registers the request and sends it clockwise along the ring. When it reaches the host, it takes on the responsibility of carrying out the agreement and join phases of the join in a manner similar to the departure processing. It makes the incoming member its monitored neighbor and delivers local membership view, view number, and other related information to it.

Both, departure and join processing must deal with the possibility of changes to membership during the agreement and commit phases. These are explained using the following definitions.

2.3 Definitions

Each member maintains a set containing all the operational members corresponding to its current group view. In addition, each member maintains a status table which stores the perceived state of all the members that are in the process of departing or joining. This table is used by a member to reject any duplicate tokens generated due to the departure of a member in the ring in the middle of any phase. There is a pool of all the tokens received by a member wherein all the tokens transferred to the neighbor are stored until removed by the update policy described later. This pool is maintained in the order of receipt and is managed so that no token is lost upon the failure of a member. Using the current group view and the status table, each member determines the member it must monitor.

Group Membership Problem: Every member, p_i, associates an integer, vn, with its current group view, denoted by the set $GV_{vn}(p_i)$, and increments it by one for every view change committed. Solution of the group membership problem requires that

$$\forall\, p_j \in GV_{vn}(p_i)\ and\ \forall\, n \leq vn, GV_n(p_j) = GV_n(p_i)$$

A GMP is safe if it guarantees the above. In the following, unless necessitated by the context, the view number will be dropped as a subscript.

Logical Ring: Assume a set of members, $GV = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$. A circular sequence of these members regardless of their physical interconnection is called a *logical* ring.

Members along the ring can be visited by traversing it either clockwise or counter-clockwise. Given such a ring, a direction of traversing it, and a member, say p_i, a relation between members gets defined by visiting each remaining member once along the ring, in order, and returning to p_i from the last member visited.

Ring Relation (RR): Given two members, $p_j, p_k \in GV$, $p_j \xrightarrow{p_i} p_k$ (read as p_j *is followed by* p_k *with respect to* p_i) if p_k is visited after p_j when starting from p_i.

Clearly, given a ring and a direction of traversal, such a relation can be defined with respect to every member in GV. On the other hand, given the above ring relation for any p_i, the logical ring has a *ring property*.

Ring Property:

$$\forall\, p_i, p_j, p_k \in GV\ if\ p_j \xrightarrow{p_i} p_k,\ then\ p_k \xrightarrow{p_j} p_i\ and\ p_i \xrightarrow{p_k} p_j$$

Every member orders its own group view as a logical ring with the above property. For a logical ring, a hypothetical marker fixed along the ring is defined.

Logical Marker: A logical marker is an fixed imaginary position between some pair of members along a logical ring.

Its adjacent members may change due to departures and joins.

Ring Host: p_{host} is the first operational member clockwise from the logical marker.

Every member p_i keeps track of the position of the logical marker by ordering $GV(p_i)$ as a logical ring with respect to p_{host}.

Rank: $rank_{p_i}(p_j)$, of any $p_j \in GV(p_i)$ is defined as the number of members between p_{host} and itself with $rank_{p_i}(p_{host})$ defined to be 0.

Monitoring Member: Every p_i maintains $p_{mon}(i)$ as the last member to query it for its health.

2.3.1 Tokens

The proposed GMP is based on circulation of three types of tokens to achieve agreement and consistent commit among members. The agreement token initiated at p_i for p_j perceived to have departed or joined is denoted as $agree_{p_i}(p_j)$. Similarly, the commit token initiated at p_i for p_j perceived to have departed or joined is denoted as $commit_{p_i}(p_j)$. Every token carries information about whether it is for a departure or join.

When a join request is received by a member other than the host, the member creates a join request token, $joinreq_{p_i}(p_j)$, and passes it on to its clockwise neighbor. When the host receives it, it generates and circulates the agreement and commit tokens for the join. If the host is the first member to receive the join request, it generates the agreement token directly.

It should be noted that the initiators of the agreement and commit tokens for a given change need not be identical and also need not be the same as the members that perceived the changes in the first place. It is possible that p_2 might perceive the failure of its neighbor p_1 and, before initiating the agreement phase, might itself fail. Then its neighbor p_3 would first initiate agreement processing for the p_2 and then initiate agreement for p_1. If p_3 fails before the agreement phase is complete then its neighbor p_4 would commit the failure of p_1, p_2 and p_3.

Every member p_i maintains a local status table, denoted as ST_{p_i}. A member has an entry in this table at p_i *only if* it has been perceived to have departed but not yet committed out of $GV(p_i)$ or if it is perceived to have joined but is not yet committed into $GV(p_i)$. This property is crucial to the safety of the protocol. The five possible values of $ST_{p_i}(p_j)$ are: *DepartureAgreed*, *JoinAgreed*, *DeparturePending*, *JoinRequested*, and *JoinPending*. The *pending* status is used to delay the committing of a change at a particular member so that

Table 1: Interpretation of $ST_{p_i}(p_j)$

DepartureAgreed	Agreement token for departure of p_j received, but it is not committed, $p_j \in GV_{p_i}$ is true
JoinAgreed	same as above, for a join, $p_j \notin GV_{p_i}$ is true
DeparturePending	Commit token for departure of p_j received, but it is not processed, $p_j \in GV_{p_i}$ is true
JoinPending	Commit token for join of p_j received, but it is not processed, $p_j \notin GV_{p_i}$ is true
JoinRequested	p_i has seen the join request from p_j on it's way to the host

the order of changes at all the operational members is identical. The rank of a member is used to determine if this status should be assigned to a member at the time the commit token for it has been received. Their interpretation is summarized in Table 1.

Every member p_i maintains a pool of all the tokens it receives, denoted as *TknPool*(p_i), in the order they are received. Tokens from this pool are deleted carefully because the receiver of a token may depart before receiving it or immediately after receiving it and the token is likely to get lost. To prevent such loss, the principle followed in token deletion is to retain a token at a member until it is guaranteed that its use is complete. The token pool update policy is described in the next section.

2.3.2 Neighbor and Host Computation

The following rules determine $p_{host}(p_i)$, the clockwise neighbor $cwnbr(p_i)$, and the counter-clockwise neighbor $acwnbr(p_i)$ using the ring relation on $GV(p_i)$ and the status table ST_{p_i}.

Rule to determine a new p_{host}: At p_i, $p_{host} = p_j \in GV(p_i)$ such that $\forall\ p_k(\neq p_j) \in GV(p_i)$, $p_j \overset{p_{old}}{\rightarrow} p_k$ where p_{old} is the old host.

This rule assigns the operational clockwise neighbor of p_{old} as the new p_{host} and is invoked to compute the new host every time a member commits the departure of its p_{host}. It should be noted that selection of the new host is determined **only** by the current $GV(p_i)$ and not along with ST_{p_i}. Since all the group views are consistent, this ensures that all the members arrive at the same p_{host}. This rule is applied whenever there is a removal of a member committed.

Rule to determine $cwnbr(p_i)$**:** The clockwise neighbor is always the member from whom the status query is received *i.e.*, $cwnbr(p_i) = p_{mon}$.

This rule is is applied whenever status query comes from a member other than the current *cwnbr*.

Rule to determine $acwnbr(p_i)$**:** $acwnbr(p_i) = p_j \in GV(p_i)$ such that $\forall\ p_k(\neq p_j) \in GV(p_i)$ $p_k \xrightarrow{p_i} p_j$ **and** $p_j \notin ST_{p_i}$.

This rule is applied whenever a timeout on the arrival of status report from the current *acwnbr* occurs and when there is a departure or join being committed.

Exception: If $p_j = p_{host}$ and $\exists\ a\ p_j$ such that $ST_{p_i}(p_j)$ changes from *JoinAgreed* to *JoinPending* or gets committed, $acwnbr(p_i) = p_j$. Upon a join, this ensures that p_{host} determines the correct member to monitor.

3 The Group Membership Protocol

In Fig. 2, the interaction of the GMP with the application and the network is shown. The network is abstracted as a set of reliable FIFO channels. The application generates the requests to join a particular group or requests the current view of a group it is a member of. In case a group already exists, the GMP has the ability to obtain the address of the nearest site with a member of the requested group running. If no site with the group is found running, it starts a new group.

Generation of a join request results in an instance of the GMP being started on the application site. This instance acquires the membership of the desired group and maintains the view information until the member departs from the group. The status change detection, agreement phase, and commit phase are described below.

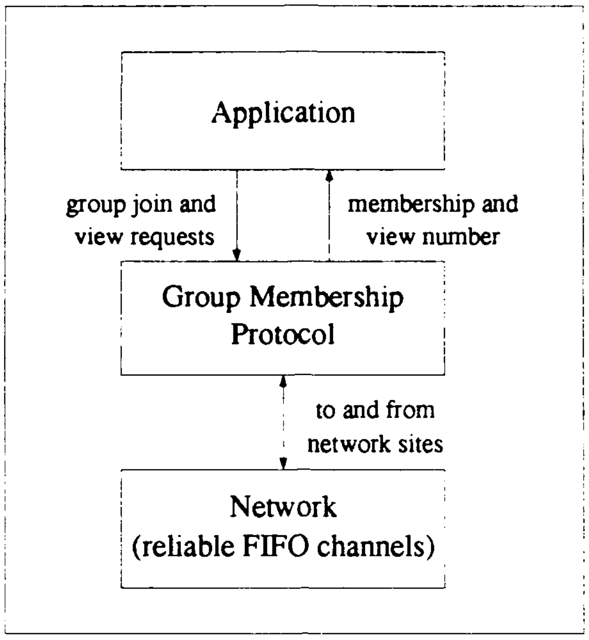
Application
group join and view requests
membership and view number
Group Membership Protocol
to and from network sites
Network (reliable FIFO channels)

Figure 2: GMP interaction with the external world

3.1 Status Change Detection

Figure 3 shows the algorithm each member executes to monitor its anticlockwise neighbor and initiate an agreement token if a departure is detected. The `Monitor` process is triggered by the local clock. The clockwise and anticlockwise neighbors are computed according to the rules given earlier in every iteration of the **while** loop. If a status message is not received, it shuts off communication with the member perceived to have departed (to prevent receipt of an excessively delayed response), updates the local status table, generates and adds an agreement token to the local pool of tokens, and sends it to the clockwise neighbor.

If this member turns out to have already departed, the status reporting instrument shown in Fig. 4 ensures that the token will get sent to the next clockwise operational member. When a change in the querying member is detected, the token pool gets sent to the new querying member in addition to the status response. It recognizes a change in the querying member by inspecting p_{mon} to send its token pool. `ReportStatus` does not compute the clockwise neighbor, but simply responds to the sender of the query.

```
Monitor process at $p_i$
1  while (true)
2      send status query to $acwnbr(p_i)$;
3      wait for $T_{pad}$; /*local timeout interval*/
4      if (status message not received)
5          shut off communication with $acwnbr(p_i)$;
6          $ST_{p_i}(acwnbr(p_i)) \leftarrow DepartureAgreed$;
7          generate $agree_{p_i}(acwnbr(p_i))$;
8          add $agree_{p_i}(p_j)$ to $TknPool(p_i)$;
9          send $agree_{p_i}(acwnbr(p_i))$ to $cwnbr(p_i)$;
10     else
11         wait for $T_{query\ period}$;
12     end if;
13 end while;
end Monitor.
```

Figure 3: Protocol for Monitoring and Agreement Initiation

```
ReportStatus process at $p_i$
1  if (querying member $\neq p_{mon}$)
2      send $TknPool(p_i)$ to the querying member;
3      $p_{mon}$ = querying member;
4  end if;
5  send status to $p_{mon}$;
end ReportStatus.
```

Figure 4: Protocol for Reporting the Status

```
InitiateJoin for a request message/token for $p_{new}$ at $p_i$
1   while (true)
2       receive join request message or $joinreq$ token for $p_{new}$
2.1      until ($p_{new} \notin ST_{p_i}$);
3       if ($p_{host} = p_i$)
4           generate $agree_{p_i}(p_{new})$;
5           $ST_{p_i}(p_{new}) \leftarrow JoinAgreed$;
6           add $agree_{p_i}(p_{new})$ to $TknPool(p_i)$;
7           send $agree_{p_i}(p_{new})$ to $cwnbr(p_i)$;
8       else
9           $ST_{p_i}(p_{new}) \leftarrow JoinRequested$;
10          add $joinreq_{p_i}(p_{new})$ to $TknPool(p_i)$;
11           if (join request) /*$p_{new}$ contacts $p_i$ first*/
12               generate $joinreq_{p_i}(p_{new})$ token;
13           send $joinreq$ token to $cwnbr$;
14      end if;
15  end while;
end InitiateJoin.
```

Figure 5: Algorithm to initiate a join

When the application generates a request to join a group, an instance of the GMP gets spawned. It obtains the address of the nearest site running a member and sends a join request message to it and waits for an intimation of the request approval for a preset interval before resending the request. Before the request is resent, the nearest site address running a member is searched again. The receiving member p_i runs an algorithm as specified in Fig. 5. A non-host member, receiving a request message for the first time generates $joinreq_{p_i}(p_{new}$ token and adds it to the local token pool. It enters status *JoinRequested* for p_{new} in its status table and sends the token to its *cwnbr*. A duplicate join request is rejected on the basis of an entry for p_{new} in the local status table. If the member receiving the request message or token is the ring host, it generates the agreement token, updates the local status table and token pool, and sends it to its *cwnbr*.

3.2 The Agreement Phase

The algorithm used to process an agreement token is shown in Fig. 6. If the member that receives an agreement token for the first time is not its initiator, it must simply pass it on to its clockwise neighbor after adding it to its token pool and updating the local status table (lines 15-19 of Fig. 6). However, if it *is* the initiator of the token, it must generate a *commit* token when the token has circulated back to it. Receiver of an *agree* token must also generate a *commit* token if the initiator had departed after generating the agreement token, and as a result, a duplicate agreement token is received at a member. In this case, the member generating the commit token will have an entry in its local status table for the initiator of the token (line 1, Fig. 6).

Any member commits a change to its view when it processes a commit token for the change. Thus the initiator of a commit token commits the corresponding change locally and sends it to the clockwise neighbor. There are two aspects to committing a change in the group view in this protocol. Firstly, since the ring configuration may lead to the arrival orders of two commit tokens to be opposite at two different members along the ring, the changes must be committed in a consistent order at all the members. Secondly, when a change is committed, it must be ensured that all the protocol-related entities are correctly updated.

The correct ordering of all changes is based on the rank of the member whose status change is being processed. The ordering is imposed at the initiator of the commit token as follows: if the rank of the member with the changed status is the lowest among all the members for which there is an agreement token in the token pool, a commit token is generated. Otherwise, commit token generation is kept pending until all changes for members with a higher rank have been committed (lines 5-13, Fig. 6).

Update of all the protocol-related quantities upon committing a change are encapsulated as **CommitChange**, whose steps are shown in Fig. 7. Aside from passing the token on to the clockwise neighbor, the local membership, view number, status table, and token pool must be updated. Line 5 determines the token pool update policy that garbage-collects old commit tokens. The principle followed in this update is that a token should be deleted from the *TknPool* only when the member is certain that its use is over. A member keeps its token pool ordered according to their arrival times, inspects all the tokens in it, and deletes all the

```
ProcessAgreementTkn for $agree_{p_j}(p_k)$ at $p_i$
    /*A commit must be generated either when I am the
    agreement initiator or when a duplicate token is received
    due to departure of the agreement initiator $p_j$*/
    if (($p_i = p_j$) || (($p_j \neq p_i$) && (duplicate token) && ($p_j \in ST_{p_i}$)))
        if (no unprocessed agreement token in TknPool)
            generate $commit_{p_i}(p_k)$;
            CommitChange;
        else
            compute rank $\forall p_l \in ST_{p_i}$ with Agreed status;
            if (rank($p_k$) is smallest)
                generate $commit_{p_i}(p_k)$;
                CommitChange;
            else
                /*depending upon whether for join or departure of $p_k$*/
                $ST_{p_i}(p_k) \leftarrow$ DeparturePending or JoinPending;
            end if;
        end if;
    else
        if ((($p_j \neq p_i$) && (not a duplicate $agree_{p_j}(p_k)$)
            add $agree_{p_j}(p_k)$ to TknPool;
            $ST_{p_i}(p_k) \leftarrow$ DepartureAgreed or JoinAgreed;
            send $agree_{p_j}(p_k)$ to $cwnbr(p_i)$;
        end if;
    end if;
end ProcessAgreementTkn.
```

Figure 6: Protocol for Agreement Tokens

```
CommitChange for $commit_{p_j}(p_k)$ at $p_i$
    /*Depending on whether a join or departure*/
1   add or delete $p_k$ from $GV(p_i)$;
2   delete $p_k$ entry from $ST_{p_i}$;
3   $vn(p_i) \leftarrow vn(p_i) + 1$;
4   send $commit_{p_j}(p_k)$ to $cwnbr(p_i)$;
5   delete all commit tokens received before
        $agree_{p_j}(p_k)$ from $TknPool(p_i)$;
6   if join committed delete $joinreq_{p_j}(p_k)$;
7   delete $agree_{p_j}(p_k)$;
8   add $commit_{p_j}(p_k)$ to $TknPool(p_i)$;
9   determine new $p_{host}$;
10  if (($join$ committed) && ($p_{host} = p_i$))
11      update $acwnbr(p_i)$;
12      send $ST_{p_i}$, $TknPool(p_i)$, and $GV(p_i)$ to $acwnbr(p_i)$;
13  end if;
end CommitChange.
```

Figure 7: Protocol for Committing a Change

commit tokens received before the agreement token for the change committed. The commit token just processed is not deleted in case the member it is sent to departs before receiving it.

If the member committing a join is the host, it updates the anticlockwise neighbor to be the new member and sends the local state to it (lines 11-12, Fig. 7). It also determines a new host (line 9), p_{host} for the ring according to the rule given at the end of section 2.

3.3 The Commit Phase

The processing of a commit token as it circulates around the ring is shown in Fig. 8. If a member is the commit initiator (*i.e.*, the token has circulated back) or if the commit token is received again, it simply exits. This indicates completion of the processing required at all members for that particular change. If it is received for the first time at a member, appropriate commit action must take place (line 4, Fig. 8). After committing the change

```
ProcessCommitTkn for $commit_{p_j}(p_k)$ at $p_i$
1  if (($p_i = p_j$) || (duplicate))
2      exit;
3  else
4      CommitChange;
5      while ($\exists\, p_l \in ST_{p_i}$ with a higher rank & pending status
           received before $agree_{p_j}(p_k)$)
6          CommitChange;
7      end while;
8  end if;
end ProcessCommitTkn.
```

Figure 8: Protocol to process a commit token

specified in this token, it is likely that a change for which a commit token generation was kept pending locally, can now be committed and propagated because it now has the lowest rank. All such pending changes can now be processed (lines 5-7, Fig. 8).

3.4 Ensuring an Identical Sequence of Commits

As members perceive departures/joins around the ring, they initiate agreement phases independently. Therefore, in this protocol, it is possible for multiple agreement phases to proceed simultaneously around the ring resulting in multiple commit tokens that circulate around the ring at the same time. The two changes divide the ring in two pieces. Clearly, the order in which these commits reach the members in these two pieces will be opposite. An identical order is maintained in this situation, as specified by lines (2 - 12) of Fig. 6.

When a commit token is to be generated, it is first checked to see if there are any unprocessed agreement tokens in the token pool. If there are, commits resulting from these are ordered identically around the ring; otherwise, a commit token is generated and change committed (lines 3 - 4). If there are unprocessed agreement tokens in the token pool, the commit initiator determines if the member for which a commit is to be initiated has the *smallest* rank among all the members for which there are unprocessed agreement tokens (lines 6 - 9). Agreement tokens for joins in the pool do not matter because members always join with the highest rank.

It should be remembered that the rank of a member is its distance from p_{host} in the clockwise direction. If the rank is not the smallest, the local status is marked as pending (line 11) and the change is committed and propagated at a later time. Thus, use of the rank ensures that all the members commit in the same order around the ring. It should be noted that the pending status for a change gets marked *only* in the commit initiator.

4 Proof of Correctness

Proposition 1: *No tokens are lost if a member updates its* TknPool *using* `CommitChange`.
Proof: If p_i receives $commit_{p_j}(p_k)$, it is guaranteed to have received $agree_{p_j}(p_k)$ some time previously because the agreement phase is followed by the commit phase. Obviously, $agree_{p_j}(p_k)$ has circulated completely around the ring. Suppose $\exists$ a $commit_{p_l}(p_m)$ received at p_i before $agree_{p_j}(p_k)$. Thus, in between the arrivals of $commit_{p_l}(p_m)$ and $commit_{p_j}(p_k)$ at p_i, $\exists$ a token, *viz.* $agree_{p_j}(p_k)$, that has circulated around the ring completely. This implies that, due to the FIFO property of channels, $commit_{p_l}(p_m)$ has circulated around the ring completely also, regardless of the locations of p_i, p_j, and p_l around the ring. Thus, $commit_{p_l}(p_m)$ has served its purpose and can be deleted from the *TknPool* at p_i. Therefore, both, $agree_{p_j}(p_k)$ and $commit_{p_l}(p_m)$ have completed their use and can be deleted. By adding $commit_{p_j}(p_k)$ to the *TknPool* at p_i, its update is complete. Since this token pool is sent to the $cwnbr(p_i)$ according to `ReportStatus`, tokens are never lost. ∎

Proposition 2: *Exactly one p_i determines itself to be p_{host}.*
Proof: `CommtChange` determines a host only when it commits a departure for the current p_{host}. According to the rule for determining the new host, only the local group view is inspected and the clockwise neighbor of the departed host is determined to be new p_{host}. According to Proposition 1, no tokens are lost. Therefore, the commit token for the departure of the old host is processed by every member. Since the host had *rank* 0, which is always the lowest, every member determines the same member as the new p_{host}. ∎

Proposition 3: *An agreement phase is always started.*

Proof: In case of a departure perceived by a member, say p_i, it may itself depart before initiating the agreement token or after sending it. In the latter case, the commit phase is carried out by $cwnbr(p_i)$. In the former case, $cwnbr(p_i)$ perceives the departure of p_i and initiates an agreement phase. It attempts to monitor $acwnbr(p_i)$ whose agreement p_i could not initiate. $cwnbr(p_i)$ perceives $acwnbr(p_i)$ as departed also and initiates an agreement phase for it. This sequence of events is extended if there is a string of departures. Therefore, the agreement phase for a departure is always started.

In case of a join, if p_i is the host and fails before initiating the agreement phase for a join, $cwnbr(p_i)$ determines itself to be the new host and receives the *joinreq* token as part of the *TknPool* to initiate the agreement phase. Since tokens are never lost, once a join request has been received by an operational member, an agreement phase for its join is always started.

■

Proposition 4: *The joining member and p_{host} behave consistently after the agreement initiation.*

Proof: p_{host} sends its GV, ST, $TknPool$, and vn to the joining member p_{new}. The exception to the rule to compute the $acwnbr$ ensures that the logical ring is correctly configured with p_{new} as the highest rank member. When the $acwnbr(p_{host})$ before the join notices that the querying member is different from its p_{mon}, it becomes aware of the new member in the ring and sends its *TknPool* to it. Therefore, all tokens that are passed to p_{host} while the state transfer to p_{new} is taking place are sent to p_{new}. This ensures that p_{new} behaves consistently with p_{host}. ■

Theorem 1: *The proposed protocol correctly solves the GMP stated as*

$$\forall\, p_i \;\in\; GV_{vn}(p_j)\ and\, \forall\, n \le vn,\, GV_n(p_j) = GV_n(p_i)$$

given that all members start with the same initial group view (GV_0).

Proof: We provide a proof by induction.

Base Case: $\forall p_i, p_j \in GV_0(p_k)$, $GV_0(p_i) = GV_0(p_j)$ at system initialization.

Induction Hypothesis: Assume that $\exists k > 1 \;\in N$ such that $\forall p_i, p_j \in GV_k(p_j)$ $GV_k(p_i) = GV_k(p_j)$.

We now prove that the next change committed by any two members is identical. Consider any $p_i, p_j \in GV_{k+1}(p_j)$. Without loss of generality, let $commit_{p_k}(p_l)$ be the next change to be committed by p_j. There are two cases.
Case 1 - $p_j \xrightarrow{p_k} p_i$: It is clear from the change detection instruments that $p_j \xrightarrow{p_k} p_l$ and $p_i \xrightarrow{p_k} p_l$. Therefore, if a change involving p_l is view change $(k+1)$ committed at p_j, either the only agreement token p_k has at the time of initiating $commit_{p_k}(p_l)$ is for p_l or p_l has the smallest rank among all agreement tokens in the *TknPool* at p_k. Now, a commit token initiated for p_m such that $p_m \xrightarrow{p_j} p_i$ cannot result in view change $(k+1)$ at p_i because this implies that p_m has a lower rank at p_i than p_l whose agreement token will be part of the *TknPool* at p_i. Therefore, agreement token for p_m would also be part of the *TknPool* at p_k and would have the smallest rank at the time of initiation of $commit_{p_k}(p_l)$. This contradicts the fact that p_l had the smallest rank at p_k or was the only agreement token at p_j. Therefore, view change $(k+1)$ committed at p_i is due to $commit_{p_k}(p_l)$.
Case 2 - $p_i \xrightarrow{p_k} p_j$: In this case, $commit_{p_k}(p_l)$ that results in view change $(k+1)$ at p_j must first pass through p_i since $p_i \xrightarrow{p_k} p_j$ and tokens circulate in the clockwise direction. This implies that view change $(k+1)$ at p_i is also due to $commit_{p_k}(p_l)$.
Thus, given the induction hypothesis for view change k, we prove that

$$\forall p_i, p_j \in GV_{k+1}(p_j)\ GV_{k+1}(p_i) = GV_{k+1}(p_j)$$

This completes the proof by induction. ∎

5 Concluding Remarks

In this report, a group membership protocol for maintaining membership information required by virtually synchronous process group based computation is described. It tolerates continuous changes to the membership by ordering the members of a group using the concept of a logical ring. In this protocol, identical processing is required to process joins as well as departures. The change detection responsibility is evenly distributed among all the members. This enables elimination of any need for centralized responsibility. By ordering all commits according to the rank of a member as defined by its position in the logical ring, the protocol correctness has been proven.

This protocol does not make any majority-based decisions. Any number of departures can occur and yet the protocol is able to function. Joins and departures can be interleaved since they are processed identically. Since there is no centralized responsibility, the overhead for committing a change is constant at $2n$, where n is the number of point-to-point messages. No special facilities such as broadcast messages, ordered access, synchronized actions are required. The protocol simply exploits the reliable FIFO nature of the channels among members. The message overhead is superior to [RB91] which is the only other group membership protocol that uses a fully connected network of FIFO channels that the authors are aware of.

Currently, this protocol is being implemented on a local area network (Ethernet) of SUN workstations using the *transport layer interface* of SunOS (a Unix variant). Objectives of the current work are to characterize the performance of this protocol in terms of the latency of a committing a change, the number of changes supported per second, and a comparative evaluation of the impact of this protocol on application level multicasts. Complete connectivity among members implies that the network is never partitioned. If the distributed computation built over this protocol spans a wide area communication network, this assumption must be relaxed. While a correctness proof has been provided here, the current work is also aimed at providing a rigorous mathematical proof.

References

[B+90] Kenneth P. Birman et al. *ISIS - A Distributed Programming Environment (Programmer's Manual)*. Department of Computer Science, Cornell University, August 1990. Rev. 2.1.

[BJ87] K. P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, pages 47–76, 1987.

[Bru85] S. A. Bruso. A failure detection and notification protocol for distributed computing systems. In *Proseedings IEEE conference on Distributed Computing Systems*, pages 116–123, 1985.

[BSS91] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, pages 272–314, 1991.

[CM84] J.-M. Chang and N. F. Maxemchuk. Reliable broadcast protocol. *ACM Transactions on Computer Systems*, pages 251-273, 1984.

[Cri88] F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proceedings of the 18th International Conference on Fault Tolerant Computing, Tokyo, Japan*, pages 206–211, 1988.

[EdL90] Paul D. Ezhilselvan and Rogerio de Lemos. A robust group membership algorithm for distributed real-time systems. In *Proceedings Real-Time Systems Symposium*, pages 173–179, 1990.

[FLP85] M. J. Fisher, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. pages 374–382, 1985.

[KGR89] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distrinuted real-time system. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications, Santa Barbara, CA*, pages 167–174, 1989.

[LSA91] L.E.Moser, P.M.Melliar Smith, and V. Agrawala. Membership algorithm for asynchronous distributed systems. In *Proceedings of the Eleventh International Symposium on Distributed Computing Systems*, 1991.

[RB91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 341–353, August 1991. Also available as TR91-1188, Dept. of Computer Science, Cornell Univ.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 52, Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5004

4. Prof. Shridhar B. Shukla 1
   Code EC/Sh, Naval Postgraduate School
   Monterey, CA 93943-5004

5. Mr. Devalla Raghuram 1
   Scientist C, Defense Electronics Research Laboratory
   Chandrayangutta Lines, Chandrayangutta
   Hyderabad, INDIA 500005

6. Director, Directorate of Training and Sponsored Research 1
   Defense Research and Development Organization
   Ministry of Defense
   227 'B' Block . Sena Bhavan
   New Delhi, INDIA 110011

7. Scientific Adviser to Raksha Mantri 1
   Director General, Defense Research and Development Organization
   Ministry of Defense
   South Block
   New Delhi, INDIA 110011

8. Director 1
   Defense Electronics Research Laboratory
   Chandrayangutta Lines, Chandrayangutta
   Hyderabad, INDIA 500005