AD-A257 442

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
ELECTE
NOV 23 1992
S
A
D

THESIS

DESIGN AND IMPLEMENTATION
OF A
GROUP MEMBERSHIP PROTOCOL

by

DEVALLA RAGHURAM

September, 1992

| Thesis Advisor: | Shridhar B. Shukla |
| Second Reader: | Douglas J. Fouts |

Approved for public release; distribution is unlimited

92-29916

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)			5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION EW Academic Group Naval Postgraduate School		6b. OFFICE SYMBOL (if applicable) 3A	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School			
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943			7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		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)

DESIGN AND IMPLEMENTATION OF A GROUP MEMBERSHIP PROTOCOL

12. PERSONAL AUTHOR(S)

Raghuram Devalla

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 09/90 TO 09/92	14. DATE OF REPORT (Year, Month, Day) September 1992	15. PAGE COUNT 127

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

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB--GROUP	Distributed processing, Group membership problem, Process groups.

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 only 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 thesis, a membership protocol based on ordering of group members in a logical ring that eliminates the need for such centralized responsibility is presented. Agreement and commit actions are token-based and the protocol ensures that no tokens are lost or duplicated due to changes in membership. It is able to process continuous changes to the membership, does not depend upon any majority-based decisions, and processes joins and departures identically. The cost of committing a change is always $2n$ point-to-point messages over FIFO channels where n is the group size. The protocol correctness is proven in a formal framework. The implementation details for the protocol to execute on a network of SUN workstations is presented. Detailed examples of the behavior of the protocol for various sequences of changes to group membership is presented. The programs for various client-server communication patterns used for interfacing various functions are also presented.

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

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
Previous editions are obsolete   UNCLASSIFIED

i

Design and Implementation
of a
Group Membership Protocol

by

Raghuram Devalla
Scientist, DoD India
B.E, Indian Institute of Science. 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the
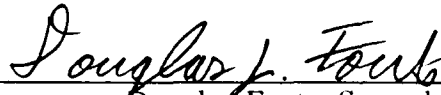
NAVAL POSTGRADUATE SCHOOL

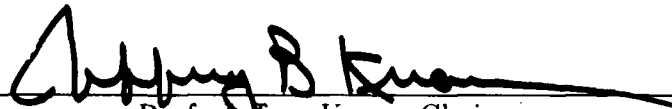September 1992

Author:

_____
Raghuram Devalla

Approved by:

_____
Shridhar B. Shukla, Thesis Advisor

_____
Douglas Fouts, Second Reader

_____
Prof. Jeffery Knorr, Chairman
Electronic Warfare Academic Group

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 only 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 thesis, a membership protocol based on ordering of group members in a logical ring that eliminates the need for such centralized responsibility is presented. Agreement and commit actions are token-based and the protocol ensures that no tokens are lost or duplicated due to changes in membership. It is able to process continuous changes to the membership, does not depend upon any majority-based decisions, and processes joins and departures identically. The cost of committing a change is always $2n$ point-to-point messages over FIFO channels where n is the group size. The protocol correctness is proven in a formal framework. The implementation details for the protocol to execute on a network of SUN workstations are presented. Detailed examples of the behavior of the protocol for various sequences of changes to group membership is presented. The programs for various client-server communication patterns used for interfacing various functions are also presented.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I. INTRODUCTION

A. BACKGROUND

Distributed computing systems are becoming increasingly popular to tackle large computational problems associated with large defense systems. A distributed system is a system with many processing elements and storage devices connected together by a network. *Fault tolerance* and *parallelism* are the two important properties of distributed systems [Ref. DSS1]. The fault tolerance capability of a distributed system is due to the replication of data and programs among several processing elements. When one processor fails, another can take over the work and complete it. The presence of several processing elements makes it possible to divide a program into several segments to be executed in parallel, resulting in a speed-up.

Exploiting parallelism or achieving fault tolerance require communication among processors. In fact, multiple processes in one processor have to communicate with multiple processes in other processors. Conventional operating systems provide a network level interface for this type of communication. In a distributed system, where such communication is basic to programs, it becomes a very complex task to manage communication between processes and write correct, efficient programs. Thus, there is a need to provide higher level communication primitives to make writing distributed programs less difficult. These primitives remove the burden of managing interprocess communication from the application developer. The important features required of these primitives are *reliability* and *minimal communication delay*.

Use of process groups is one of the approaches to write distributed applications [Ref. B+90]. It is based on reliable communication and simplifies the writing of

1

applications. Process groups occur when groups of processes cooperate to perform a task, share memory, subdivide computation, and so forth. For example, there could be a process group where the leader performs the task of searching the database and, in the event of its failure, some other member takes over and finishes the task. In this example, process groups are used to provide a fault tolerant service.

The main features of the process group approach are, *failure atomicity*, for multicasts, and ,*membership atomicity*, for failures as well as joins to a group. A failure is atomic for a multicast if all members receive a multicast or none of them receive it. *Membership atomicity* means that if a member joins or leaves the group, every one agrees on it or no one agrees to it.

This sort of a guarantee provided by reliable primitives leads to the requirement of all members in a group commiting to a failure or a joining of a member in the same sequence such that there is a consistency in the membership changes to the group at all member sites. The Group Membership Problem (GMP) is the problem of agreeing on the membership of a group and disseminating that information consistently among the members of the group.

B. OBJECTIVES OF THE STUDY

In this thesis, a decentralized mechanism for providing a consistent group view at all member sites is presented. This approach assumes that the members are interconnected over a network of reliable FIFO channels. The GMP is solved by requiring that each increment of the view number be assosiated with successive views that differ by only one member. This approach also assumes that the only failure mode is *fail-stop* [Ref. Cri88] and the processors do not behave maliciously. This approach uses tokens for achieving agreement and commit actions.

Unlike all the other approaches which are described in the next chapter, this approach is a distributed approach and hence eliminates centralizing the responsibility of ensuring consistency of view changes. This approach scales linearly with respect to the number of messages as the number of members increases. It also guarantees that the protocol is non-blocking and members can leave and join the group continuously.

C. THESIS ORGANIZATION

This thesis has six chapters. The second chapter deals with the earlier membership protocols, their merits, and problems.

In Chapter III, the decentralized membership protocol is discussed and formal algorithms are given. The implementation details, details of the process specifications, and the data structures used are given in Chapter IV. Since the implementation involves a lot of interaction among various processes, the system calls used in the implementation are also discussed in this chapter. Chapter V gives an extended example of the working of the protocol. Chapter VI deals with the future work in this area. Appendix A gives a listing of the programs developed.

II. EXISTING APPROACHES

This chapter deals with the existing approaches to the Group Membership Problem (GMP). It first describes GMP and how it is useful in Electronic waifare applications. It then describes, in detail, various approaches to GMP.

A. GROUP MEMBERSHIP PROBLEM

The task of managing a distributed computation containing replicated processes is best formulated in terms of management of process groups, where each group represents a fault-tolerant process. The process group's membership changes when its processes fail (they are removed), when they recover (they are re-instated), when new processes join, and when processes leave voluntarily. The process group's members query the membership view and are able to take actions based on the membership view. Agreement on the membership of a group of processes is a must to avoid inconsistency problems. This problem of agreement on the membership of the group is defined as the Group Membership Problem.

1. Importance

For example, let us consider an Electronic Warfare system that is required to perform a complicated task of countermeasures initiation based on threat assessment. Threat assessment is based on several parameters like the type of enemy platform, threat priority, and the most effective countermeasures possible. These parameters are evaluated by a group of processors which interact with one another through messages. The messages could be broadcasts, i.e., the same message is sent to different processors for a collective action. It is necessary that all operational processors agree on the failures to take correct and consistent corrective action.

4

Let us consider that a processor A sends messages to two other processors, B and C, for a particular sequence of actions to take place. If A fails after sending the message to B, and before sending the message to C, it is possible that B does not know of the failure of A. In this case, the action taken by B and C could be inconsistent and erroneous. If both, B and C knew of the failure of A then they would be able to recover from the failure based on consistent information. Similar examples can be found in the database field and in real time applications. [Ref. CT90]

2. System Classification

Distributed systems can be classified into *synchronous* and *asynchronous* systems. In synchronous systems, all events are deemed to happen one at a time. In this type of system the groupview is frozen at the time of message sending. All messages wait till the changes to group membership are complete and all membership changes wait till all pending messages are sent. There is a close synchronization in the clocks of the interacting processes and there is a known upperbound on the message delivery time.

In an *asynchronous* system, there is no relationship between the clocks of interacting processors. The time for message delivery is unknown. It is not possible to be certain of the failure of any process, since there is no upper bound on the time a message takes to be delivered. Therfore, processes are only *perceived to have failed* and crashes are indistinguishable from communication delays. It is necessary that processes perceived to have failed be removed from the group. If not, it is impossible to reach a consensus on the failure of a processor [Ref. FLP85]. In the rest of the chapter, several existing approaches to group membership problem are discussed.

B. SYNCHRONOUS SYSTEMS

In a synchronous distributed system the processor clocks are synchronized. This clock synchronization leads to availability of a global time at all processors. There is a known upper bound on the message delivery time and this leads to detection of a failure in a bounded time. Cristain has given one method for solving the GMP [Ref. Cri88] in this setting. Another approach is given by Ezhilselvan and Lemos [Ref. EzLe90]. These are discussed in greater detail in the subsequent paragraphs.

1. Periodic Broadcast Protocol

This protocol, developed by Cristain [Ref. Cri88], assumes a synchronous communication network which provides a message diffusion service and a bounded delay on the message delivery time. These assumptions lead to the premise that two processors are unable to communicate only if one of them has failed. It also assumes an atomic broadcast tolerant of performance failures, i.e., failure of the communication link to deliver messages within a known bounded time.

In this protocol, all members periodically send messages to one another about their presence. Since an atomic broadcast is assumed, all operational processors receive this message. If a processor fails, then it is not able to send the periodic message and all the other members know of its failure within a bounded delay and remove the failed member from their view. Each of these periodic messages contains the clock time associated with it and is used to synchronize the clocks. The renewal time for the next broadcast is a constant time added to the synchronized time.

When a new processor wants to join, it sends a *new-group* message to all the members with the group id. All the members of that group respond to the joining processor by sending the *present* message to it. These messages are used by the joining processor to create its view of the group at that point. The time of renewal is now changed to take into account the member joining.

6

This protocol has the advantage that failures are detected in the quickest possible time. If a processor failed immediately after signaling its presence. its failure is noticed within time equal to the message renewal time added to the maximum message delivery time. The drawback of this protocol is that it requires n atomic broadcasts every group renewal time, where n is the number of members in the group.

2. Attendance List Protocol

This protocol was also developed by Cristain [Ref. Cri88] and assumes the same type of broadcast facility and communication network as the previous one. The joining of the new members is also handled in the same way. In this protocol, the membership is checked by sending a datagram to all the members some time after a join is completed. This datagram reaches all members within a bounded time and all members check to see if they received it within the right time. If there is a failure, at least one of them does not receive the list and it issues a new join phase. In this phase, the member which has failed does not participate and his membership is removed from the group by other members. This protocol has a reduced message overhead in the absence of changes and is more efficient than the periodic broadcast protocol. This reduced overhead leads to an increase in the failure detection time.

3. Robust Group Membership Algorithm

This algorithm, proposed by Ezhilselvan and Lemos [Ref. EzLe90], is developed for real-time systems. It assumes a failure free broadcast network which preserves the order of messages and has a bounded interval on message delivery time. It also assumes that all processors access the medium in a known order and can detect the absence of processors broadcast in a bounded time interval. This algorithm deals with send-failure, receive-failure and crash-failure.

All processors maintain a vector, denoting the status of each member, which is transmitted periodically to all members. in a particular time slot known to

all members. During each cycle, all processors exchange this information with one another. In this way, all processors have the same information and each of them, by executing the same algorithm, arrives at the same results. The member status is continually updated depending on the message received. If no message is received then its status is denoted as message absent, and if it is different, it is designated as failed. This is because the processor might have a receive-failure and might not have received the messages. Based on these updates, the new group membership is determined. All processors need to check for a majority of operational processors in a cycle. If more than the majority fail, then the status of the processor is set to failed and it stops execution.

This algorithm takes action in a distributed manner. Its drawback is that it requires all processors to have a priori knowledge of the sequence of medium access for all processors.

C. ASYNCHRONOUS SYSTEMS

In an asynchronous distributed system, there is no upper bound on the message delivery time. There is no synchronization of clocks, and hence, the concept of global time is not there. Since the delivery delay is unbounded, it is impossible to distinguish between failures and communication delays. Therfore, in an asynchronous system, processors are only perceived to have failed. Because of these constraints, the algorithms require multiple message rounds for committing a change. The following paragraphs deal with approaches by Chang, Birman, Bruso, and Smith.

1. Failure Detection and Notification Protocol

This protocol, developed by Bruso [Ref. Bru85], is aimed at distributed database systems with a token ring network. It detects the failure of nodes and notifies all nodes when a recovery is complete. It is designed for crash failures and commu-

nication isolation. In this protocol, an acknowledgement is required for all messages. The protocol is divided into failure detection and recovery reporting segments

The failure detection is decentralized and is achieved in the following way. If a processor does not receive an acknowledgement for a message after many retransmissions, it initiates the failure detection by sending node down messages to all other processors. All processors receiving this message retransmit it to all other processors. The approach is robust with respect to multiple failures. This leads to a flood of messages where each processor sends to every other processor. Each of these messages sent is acknowledged and the replies are used for determining which nodes are reachable. If an acknowledgement is not received, then the failure detection is started for the node for which acknowledgement was not received. All processors, on receiving a node down message, do not attempt to verify it and change the status to down. In this way, the integrity of the member status at all nodes is preserved.

The recovery reporting is a centralized function. When a processor recovers, it sends a node up message to all processors. Processors, on receiving the message, will note the status as up and acknowledge to the recovered process. The recovered process updates its local status based on the acknowledgement. It also verifies that all processors that it can reach also agree about the status of all down nodes. This protocol uses a version number for reaching agreement. A version number denotes the number of times a node has failed. This is done by initiating failure detection protocol for the members from which acknowledgement was not received. This causes all other processors to follow suit. The greatest merit of this system is that it is simple and robust for multiple failures. Its problem is that the number of messages does not scale linearly as the number of members increases.

2. Protocol Based on Total Message Ordering

This protocol by Moser, et.al., is built on top of protocols guaranteeing reliable and totally ordered message broadcasts[Ref. LSA91]. This protocol is non-blocking and tolerates partitions. It assumes an underlying fault tolerant ordering protocol and no broadcasts are delayed when there is a membership change.

In this protocol, all messages are associated with ordinal numbers denoting their position, since total order common to all processors is assumed. There is an agreement process running on the ordering of messages. If a processor does not order a message for a long time it is deemed to have failed and a failure notification message is sent by the processor identifying failure. Since all messages are ordered, this message is sent in the same relative order to all other members and the processor is removed from the group.

When a processor wants to join the group, it sends a special message called a request message. This message, when received by another processor, is ordered with an ordinal number. It sends an acknowledgement message to the requesting processor with the ordinal number of the most recent message it has ordered. The processor now orders all the messages it receives and, when it receives the ordered message, determines that it has been admitted to the group and starts sending messages. These protocols only give incremental changes to the configuration. For getting the complete view, it is possible to have initialization algorithms which initialize the view each time a processor joins the configuration.

This protocol has a low communication overhead for membership changes. However, it assumes a total order on messages which involves multiple message rounds for agreement.

10

3. Protocol Based on Rotating Token List

This protocol by Maxemchuk and Chang is developed as a result of developing reliable broadcast protocols using a rotating token site [Ref. CM84]. In this protocol, a reformation of the token list occurs whenever there is a failure or recovery. This is a three phase protocol. The protocol assumes a datagram service and assumes fail-stop behavior. There is a token list of all members in the group. A site which detects a failure or recovery is the originator of the protocol. It invites other sites in the group to form a new list. To prevent multiple lists being generated a site can join only one list, and the list containing the majority is taken to be the valid list.

All lists have a version number attached to them. A site can join only lists with a higher version number. These two rules are used to generate only one valid list. The originator receives responses from the other members with the timestamp of the next message they are expecting and the version number of the list. If the response is from a majority of the members in the old list, it creates a new list consisting of members who have responded and passes it to all the members in the new list. If it does not have a majority, it aborts the reformation phase. In phase 3, if all the members in the token list agree, the new token list is committed and the token is passed to the new token site. The token site is selected based on the timestamp of the message and the member with the largest time stamp is elected to be the token site because it has received the most number of messages. Timeouts are incorparated in the protocol such that there is no eternal wait for responses.

This protocol is a blocking protocol and is likely to be unmanagable if there are frequent changes to the group. When the token site fails, the reformation protocol is more complicated and requires another round of communication to recover any lost mesages.

4. ISIS Approach

This approach is a nonblocking approach developed by Birman, et.al., [Ref. RB91]. It assumes that processes communicate through a completely connected network of reliable FIFO channels. There is no bound on message delivery times and there is no global clock. It does not assume any underlying fault tolerant communication. The processes only fail by crashing and all recovering processes are joined as new processes.

This protocol is a centralized approach. There is a process designated as *Mgr* (for manager) which is responsible for coordinating updates to the local views of other processes. When a process finds another process faulty, it informs the *Mgr*. *Mgr* then initiates a two phase protocol to commit the failure. It sends a message informing the failure of a member to all the other members of the group and awaits their response. At the end of this phase, all operational members agree on the failure of the member. In phase 2, *Mgr* broadcasts a commit message that informs all members to remove the member from their groupview.

If the *Mgr* fails in the middle of a commit phase, no system view will exist. To reestablish the view, the reconfiguration algorithm deals with progression and succession problems. This is a three phase algorithm. The initiator broadcasts the reconfiguration interrogation message to all the members in its local view. The initiator is a member who has been the member of the group for the most number of changes. The initiator broadcasts a reconfiguration interrogation message to all processors in its local view and awaits its response. Based on the majority response, the initiator determines an update event, based on the local states of the other processes. The execution of this event restores the system view. The initiator broadcasts this as the reconfiguration proposal message. It then sends the commit message after receiving a majority response. Election of a new manager must avoid invisible commits.

The protocol tolerates only minority of failures in successive views and is one of its problems.

III. GROUP MEMBERSHIP PROTOCOL

In the previous chapter, several approaches to GMP were discussed and their merits and problems were highlighted. A decentralized Group Membership Protocol has been proposed by Shukla and Devalla in [Ref. ShDr]. In this thesis we further elaborate on the protocol and describe an implementation. This chapter describes the basic functions and algorithm of the protocol. and therfore , contains material that has been directly reproduced from [Ref. ShDr].

The basic functions required of group membership protocols are to detect changes in the membership and ensure that all operational members commit these changes to their local views consistently. The consistent commit requirement entails an agreement about the change detected. Given such a protocol, higher level tools for constructing distributed applications, such as ISIS [Ref. BSS91], can be constructed. Solution of the GMP is complicated by the following two properties of asynchronous distributed systems. Firstly, since it is impossible to distinguish a failed process from a slow process. failure detection is not possible. Any failure is only a *perceived* failure that everyone must commit to eventually. Secondly, unless the underlying network communication is embellished in some manner, such as total ordering of messages [Ref. LSA91] or total ordering of access to the communication medium [Ref. EzLe90], the consistency must be achieved using only a network of reliable, *first-in-first-out* (FIFO) communication channels, the delay over which is unbounded. This implies that agreement and consistent commits can only be achieved by multiple message rounds.

In this approach, as in [Ref. BSS91], it is assumed that all communication between members of a group carries a view number. The GMP is solved by requiring

14

that each increment of the view number be associated with successive views that differ in only one member and guaranteeing that a given view number, at any operational member, has the same membership. The protocol proposed herein uses a completely connected network of reliable FIFO channels and incorporates continuous changes to the group membership without the need for *a priori* knowledge of potential members. This approach eliminates the need for centralizing the responsibility of ensuring consistency of view changes as in [Ref. 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 tokens generated during its operation.

A. ASSUMPTIONS, OVERVIEW, AND DEFINITIONS

Our objective is to develop a group membership algorithm that can be used as the basis of fault-tolerant process group-based communication primitives such as those described in [Ref. B+90, BJ87, BSS91]. As mentioned previously, it is assumed that every membership view at a member is assigned a view number and views corresponding to successive numbers differ by exactly one change (either deletion or addition of a member). Reliable FIFO communication channels between any two processors that are operational is assumed. All failures are assumed to be *crash* or *fail-stop* [Ref. 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 to the group when a *join* request is processed and gets deleted from the group when a *departure* is perceived. It is assumed that the group

15

name is public to those processes that may wish to join the group. Some mechanism is assumed to exist whereby the process wishing to join can query the operating system at a site if it is running a member process of the known group name.

1. Overview

Group Membership Protocol (MP) guarantees that the changes to the group view and their sequence at each operational member are identical. Using a view number in all group-related communication guarantees that fault-tolerant group communication can be achieved. The principle feature of the MP is that there is no central element either to detect a member's change in membership status or to guarantee consistency of a commit action on the group membership. Both are achieved in a distributed manner using a logical ring which is simply a conceptual circular ordering of the members. It has no relation with the physical locations of the members. Given such a ring and a direction of traversing it (clockwise is selected for no particular reason), 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. 3.1. Every member periodically sends a status query and receives a message that indicates that the monitored member continues to be a group member. Initially, the ring configuration is known to all the members. As members change status, either voluntarily or involuntarily, the ring configuration changes. The protocol maintains sufficient information at each
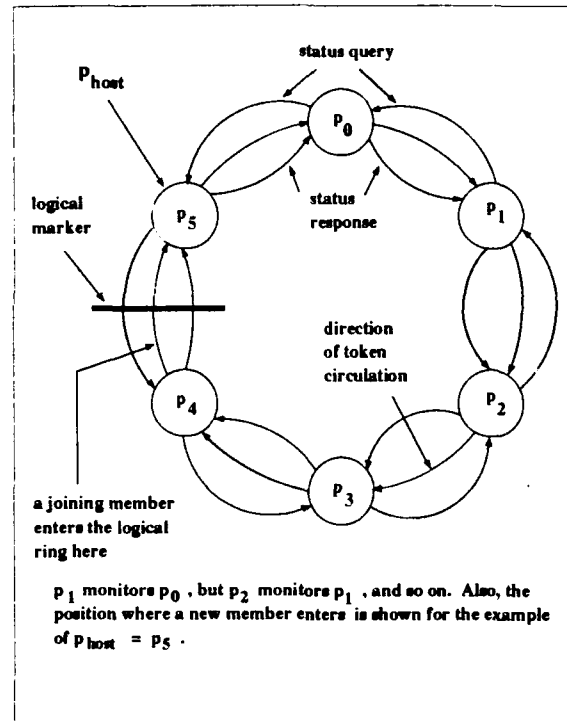
Figure 3.1: A Logical ring

operational member to enable it to determine the processor it must monitor.

The MP treats the cases of a member leaving the group in the same manner as a member joining the group. [1] When a member departs voluntarily, it simply stops responding to the status query from its monitor. Upon failure, 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 its query, the monitored member is perceived to have left the group and the algorithms to ensure that all the operational members consistently commit to this change are invoked. When a member recovers or wishes to join anew, it sends a *join* request to the first group member it can locate. The member wishing to join interrogates all the member sites which could have the group running at its site. It is assumed that the operating system at each member site has a knowledge of all the groups that the processes executing in it are members of. The

---

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

17

MP guarantees that only one of the operational members of the group processes the join request. 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 no tokens are lost due to departures. They also guarantee that the protocols are robust with respect to generation and processing of duplicate tokens.

a. Processing of Individual Changes

Simple cases of individual changes are first illustrated to orient the reader and a more detailed description is given in the next section.

A single departure is processed as follows. 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 failure perceived. 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.

A join is processed as follows. 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. 3.1, a new member always enters as the counter-clockwise neighbor of the host who has the responsibility

of carrying out the agreement and commit phases for the new member. It should be noted that identifying a member as failed merely determines who will initiate the agreement phase for a join request and does not represent the centralization of any function. Although the designation as a host may move from one member to another in the clockwise direction due the departure of the host member itself, the protocol enables all the operational members to maintain knowledge of the current host of the ring. It makes the incoming member its monitored neighbor and delivers local membership view, view number, and other related information to it.

2. Definitions

Certain items of information are maintained locally at each member to ensure the correctness of the protocol. All members maintain a set corresponding to the current group view containing all the operational members. In addition, they maintain a status table locally 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.

a. Group Membership Problem Definition

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

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

19

Therefore, a group membership protocol is correct if it guarantees the above. In the following, unless necessitated by the context, the view number will be dropped as a subscript.

b. Logical Ring

Assume a set of members. $GV = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$, forming a *logical* ring. A *logical* ring is simply a circular sequence of these members regardless of their physical interconnection. 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, we define the following relation by visiting each remaining member once along the ring, in order, and returning to p_i from the last member visited. Using this ordering of members, the following relation can be defined.

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 the following *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, we define a logical marker along the ring that does not move. However, its adjacent members may change due to departures and joins. 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} where p_{host} is the first operational member clockwise from the logical marker. Every $p_j \in GV(p_i)$ has a rank, $rank_{p_i}(p_j)$, defined as the number of members between p_{host} and itself with $rank_{p_i}(p_{host}) = 0$. Every p_i maintains p_{mon} as

the last member to query it for its health.

c. Tokens

The proposed protocol is based on token circulation to achieve agreement and consistent commit actions 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 $commit_{p_i}(p_j)$. When there is a potential member wishing to join the group and the request is received by a member of group other than the host, the member who receives it sends a join request token denoted by $joinreq_{p_i}(p_j)$. 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 token carries information about whether it is for a departure or join. 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 is an important property, since the correctness of the protocol depends upon it. The five possible entries of $ST_{p_i}(p_j)$ are: *DepartureAgreed, JoinAgreed, DeparturePending, JoinRequest,* and *JoinPending.*

DepartureAgreed entry signifies that the agreement token for the member to leave has been initiated and it is yet to be committed and removed from groupview. *JoinAgreed* entry is same as *DepartureAgreed* except that it is for join-

ing a group instead of leaving the group. *DeparturePending and JoinPending* entries signify that agreement phase is completed but there are other changes to be committed before committing this change to maintain consistency at all member sites of the order of committing the member. *JoinRequested* entry signifies that a potential member has sent a request for joining the group and that this member has passed on the information to its neighbor on the way to the host. 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. The principle followed in token deletion is to retain a token at a member until it is guaranteed that its use is complete. The *TknPool* update policy is as described in the next section.

d. 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}.

- <u>Rule to determine a new p_{host}</u>: At p_i, $p_{host} = p_j \in GV(p_i)$ *such that* $\forall$ $p_k(\neq p_j) \in GV(p_i)$,

  $p_j \xrightarrow{p_{old}} p_k$ 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}.

  <u>Time of application:</u> This rule is applied whenever there is a removal of a mem-

ber committed.

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

  Time of application: 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 $p_j \notin ST_{p_i}$.

  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.

  Time of application: This rule is applied whenever a timeout on the arrival of status report from the current $acwnbr$ and when there is a removal or join being committed.

B. THE MEMBERSHIP PROTOCOL

For a departure, the MP at a member is activated either by non-receipt of the status response from its $cwnbr$, the monitored member, or by the receipt of a departure agreement token from its $acwnbr$. In case of a join, it is activated if it is the p_{host} and receives a $JoinReceived$ token from its $acwnbr$. $JoinReceived$ token processing is described in a greater detail in chapter IV .

We shall first describe the change detection instruments of this protocol. We follow this with description of the agreement and commit algorithms executed at any member. It should be remembered that the membership view at p_i is arranged as a logical ring, and therefore, the ring relation is defined on it. Also, every member

places a logical marker on its own logical ring.

1. Status Change Detection and Agreement Initiation

Figure 3.2 shows the protocol each member executes to monitor its counter-clockwise neighbor and initiate an agreement token if a departure is detected. The **Monitor** process is triggered by the local clock. The clockwise and counter-clockwise 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 it to the local pool of tokens, and sends the agreement token to its clockwise neighbor.

Note that only an operational member that does not have an entry in the status table is determined to be the *cwnbr* by the rules.

If this member turns out to have already departed, the status reporting instrument shown in Fig. 3.3 ensures that the token will get sent to the next clockwise operational member. When a change in the querying member is detected, the *TknPool* 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 *TknPool*. It should be noted that **ReportStatus** does not compute the clockwise neighbor, but simply responds to the sender of the query.

Similarly, when a member receives a *JoinRequest*, it executes a protocol as specified in Fig. 3.4. A non-member wishing to join a group finds the nearest site running a process that is a member of the group it wants to join. It sends a join request to this member and waits for an intimation of the request approval for a preset interval before resending the request. Duplicate requests are handled as described below. The member receiving the request does the following:

```
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)) ← DepartureAgreed;
7           generate agree_{p_i}(acwnbr(p_i));
8           add agree_{p_i}(p_j) to TknPool;
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 3.2: Algorithm for monitoring and agreement initiation

- If the request is not a duplicate, it generates *JoinReceived* token with the requester's address in it. If the request is a duplicate, the member ignores it.

- Enters this token in its *TknPool*, makes an entry in ST and sends it to its *cwnbr*.

2. The Agreement Phase

The algorithm used to process an agreement token is shown in Fig. 3.5. If the member that receives an agreement token for the first time is not its initiator

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

Figure 3.3: Algorithm for reporting the status

```
InitiateJoin for a request from p_new at p_i
1   while (true)
2        read Tknpool for JoinRequests;
3        if (p_host = p_i)
4            generate agree_{p_i}(p_new);
5            ST_{p_i}(p_new) ← JoinAgreed;
6            add agree_{p_i}(p_new) to TknPool;
7            send agree_{p_i}(p_new) to cwnbr(p_i);
8        else send joinreq_{p_i}(p_new) to cwnbr;
9        end if;
10  end while;
end InitiateJoin.
```

Figure 3 4: Algorithm to initiate a join

then it must simply p:.s it on to its clockwise neighbor after adding it to its token pool and updating the local status table (lines 15-19 of Fig. 3.5). However, if it is the initiator of the token, it must generate a *commit* token. It must also generate a *commit* token, if a member receives a duplicate agreement token with an initiator that has an entry in its status table denoting the failure of the initiator.(line 1, Fig. 3.5).

The member commits a change to its view when it sees a commit token for it. Therefore, the initiator of a commit token must commit the change locally in addition to generating and sending it. There are two aspects to committing a change in the group view in this protocol. Firstly, since the ring configuration may lead to two commit tokens arriving at two different members in the opposite order, 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. All the effects of committing a change as **CommitChange** whose steps are shown in Fig. 3.6.

3. The Commit Phase

The processing of a commit token as it circulates around the ring is shown in Fig. 3.7. If the receiver is the commit initiator (token circulates back to its initiator) or if the commit token is received again, it simply exits. This indicates completion of the protocol for that particular change. If it is received for the first time at a member, appropriate commit action must take place (line 4). After committing the change 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). We now discuss the actions required for committing a change (CommitChange).

a. Effects of a Commit Action

All the effects of a commit action are shown in Fig. 3.6 as CommitChange for $commit_{p_j}(p_k)$ received at p_i. The straightforward effects are deletion of p_k from the group view at p_i, update of p_i's local status table, its view number increment, and passing the token on to p_i's clockwise neighbor. There are three other important effects that must take place when a commit token is generated. First, it must determine a new host (line 8), p_{host} for the ring according to the rule given at the end of section 2. Second, it must take appropriate action if the change committed is a join (lines 9 – 11). The additional function to be performed when committing a join is to send the current group view, view number, local status table, and the token pool to the joining member. This is essential to ensure that the new member has up-to-date, consistent information about the group at the time of joining. Receiving it from p_{host}, which is clockwise from itself, guarantees that the new member behaves consistently with the host. Finally, committing a change locally presents an opportunity to correctly update the local *TknPool* (lines 5 – 7). The principle followed in this update is that a token should be deleted from the *TknPool* only when the member is certain

27

that its use is over. It allows inspection of all the tokens in it and keeps them ordered according to their arrival. As specified by the ReportStatus process of Fig. 3.3, the entire *TknPool* at a member is sent whenever the *cwnbr* changes. This happens when a member that perceives the departure of its counter-clockwise neighbor establishes a new counter-clockwise neighbor by querying it for status. The new counter-clockwise neighbor sees a change in the member querying it, and therefore, sends its *TknPool* to the new monitor.

b. 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 two 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. 3.5. 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). [2] 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.

---

[2] Agreement tokens for joins in the pool do not matter because members always join with the highest rank.

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.

C. CORRECTNESS OF THE PROTOCOL

We prove several propositions relating to the correctness of the protocol proposed.

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.*, $agree_{p_j}(p_k)$ has circulated around the ring completely. This implies that $commit_{p_l}(p_m)$ has circulated around the ring completely also, regardless of the locations of p_i, p_j, and p_l around the ring due to the FIFO property of channels. 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. If this token pool is sent to the $cwnbr(p_i)$ according to `ReportStatus`, no tokens will be 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

```
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*/
1   if (((p_i = p_j) || ((p_j ≠ p_i) && (duplicate token) && (p_j ∈ ST_{p_i})
1.1 && (p_l ∈ ST_{p_i} ∀p_l s.t.  p_l ->^{p_j} p_i)))
2       if (no unprocessed agreement token in TknPool)
3           generate commit_{p_i}(p_k);
4           CommitChange;
5       else
6           compute rank ∀p_l ∈ ST_{p_i} with Agreed status;
7           if (rank(p_k) is smallest)
8               generate commit_{p_i}(p_k);
9               CommitChange;
10          else
                /*depending upon whether for join or departure of p_k*/
11              ST_{p_i}(p_k) ← DeparturePending or JoinPending;
12          end if;
13      end if;
14  else
15      if (((p_j ≠ p_i) && (not a duplicate agree_{p_j}(p_k))
16          add agree_{p_j}(p_k) to TknPool;
17          ST_{p_i}(p_k) ← DepartureAgreed or JoinAgreed;
18          send agree_{p_j}(p_k) to cwnbr(p_i);
19      end if;
20  end if;
end ProcessAgreementTkn.
```

Figure 3.5: 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) ← 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;
6   delete joinreq_{p_j}(p_k);
7   delete agree_{p_j}(p_k);
8   add commit_{p_j}(p_k) to TknPool;
9   determine new p_{host};
10  if ((join committed) && (p_i = p_{host}))
11      update acwnbr(p_i);
12      send ST(p_i), Tknpool(p_i) and GV(p_i) to the acwnbr(p_i);
13  end if;
end CommitChange.
```

Figure 3.6: Protocol for committing a 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 (∃ p_l ∈ 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 3.7: Protocol to process a commit token

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.

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 *JoinReceived* as part of the *TknPool* to initiate the agreement phase. Argue that no join requests are lost. ∎

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 exceptions to the rules to compute $cwnbr$ and $acwnbr$ ensure 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 it *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

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

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

Proof: We provide a proof by induction.

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

<u>Induction Hypothesis:</u> Assume that $\exists k > 1 \in \mathcal{N}$ 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.

<u>Case 1 – $p_j \overset{p_k}{\to} p_i$:</u> It is clear from the change detection instruments that $p_j \overset{p_k}{\to} p_l$ and $p_i \overset{p_k}{\to} p_l$. 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 $p_m \overset{p_j}{\to} p_i$ 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 $commit_{p_k}(p_l)$.

<u>Case 2 – $p_i \overset{p_k}{\to} p_j$:</u> 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 $p_i \overset{p_k}{\to} p_j$ 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. ∎

IV. IMPLEMENTATION OF THE PROTOCOL

In this chapter, the implementation aspects of our Group Membership Protocol(MP) are discussed. Major functionalities of the protocol are detection of failure, agreement of failure, committing of failure, addition of members, and supplying the current view to application processes. This requires the protocol to communicate with application processes executing on the same member, MP at other members, and be able to act on the information recieved from other processes. The action taken by the MP depends on the data received. The data that it receives from and sends to the external world is depicted in Fig. 4.1. Based on this data flow, the software design for the protocol was developed as a set of interacting processes each performing a unique function. The design used the utility Software Through Pictures [Ref. STP] to visualise the interactions and to check for consistency in the data exchanged. The following paragraphs give a more in-depth picture of the implementation details.

A. PROTOCOL SOFTWARE DESIGN

Fig. 4.1 gives the interaction of Group Membership Protocol with the external world.

1. Functions in the Protocol

The Group Membership Protocol will be executed at all member sites of a process group. This diagram gives the interaction of the MP with the application program executing in the same member site and the MP executing at other member sites. New Members and application requests for current group view are applications executing on the same host. Clockwise and anticlockwise neighbor represent MP execution at other member sites. The protocol receives parameters (*Token and*
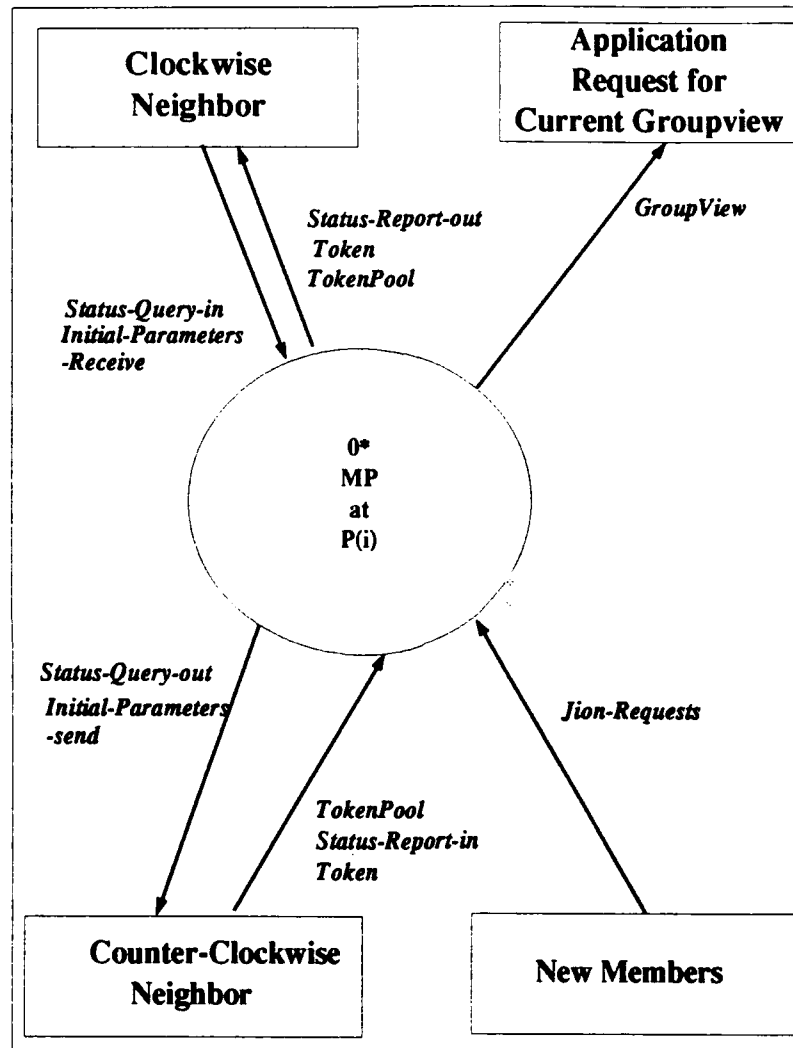
Figure 4.1: Topmost view of MP interactions

TokenPool) from other member sites and updates its membership view accordingly. It sends status query (*Status-Query-out*) to counter-clockwise neighbor and awaits the reply from it (*Status-Report-in*). It receives status query (*Status-Query-in*) from clockwise neighbor and sends reply (*Status-Report-out*) as a response. When it gets a message requesting to join the group (*Join-Requests*) it acts on it in an appropriate manner. Details are explained in the lower level descriptions. When this member site joins a new group, it receives initial parameters (*Initial-Parameters-Receive*) and initializes its parameters(*Group View, TokenPool, Status-Table*). An application wanting the current members of the group (*Group View*) queries the MP to get the current membership.

2. Subcomponents of MP

Fig. 4.2 gives the overall view of the various functions in the MP software running at every member process. The functions are implemented by a number of sub-processes. The name of the processes and their functions are given in Table 4.1. A detailed process specification is given in the lower level diagram corresponding to each process defined in this diagram.

B. DATA STRUCTURE DEFINITIONS

Different data structures used for implementation and in process specifications are described below.

Address is a special data type defined as a *long*. It is generated by the Unix system calls. It uses the conventional Internet address and the port address to generate a unique address. It is used for communication with processes spread over different hosts.

Action-message: This is defined to enunciate several action-oriented messages on stored data. They include add, remove, and update of a linked list. The defi-

37

Initial-Parameters-Receive

Join
Initial

Initial-TokenPool

Initial-
StatusTable

Initial-GroupView

TokenPool
Manager

Token

GroupView
Manager

StatusTable
Manager

1

2

Token

7

*Neighbor-
Member*

*Current-
GroupView*

*Member-
Status*

6

5

Initiate
Join

4

3

8

Agreement
Process

*Update-
Status
Table*

Commit
Process

Token

*Initiate-
Commit*

TokenPool

Token

LEGEND

1 *Update-GroupView*

2 *Update-Status*

3 *TokenStatus*

4 *Current-StatusTable*

5 *Current-TokenPool*

6 *Update-Status*

7 *Update-TokenPool*

8 *Neighbor-Status*

Initiate
Departure

*Status-
Report-out,
Status-
Query-out,
TokenPool,
Token*

*Status-Reprort-in,
Status-Query-in,*

*Initial-
Parameters-
Send*

Token

FIFO
Channel
Layer

*Status-Report-out,
Token,
TokenPool*

*Status-Query-out,
Initial-Parameters-Send*

*Status-Report-in,
Token,
TokenPool*

*Status-Query-in,
Initial-Parameters-Receive*

Figure 4.2: MP process interactions

38

TABLE 4.1: PROCESSES AND THEIR FUNCTIONS

NAME	FUNCTION
FIFO-Channel-Layer	This process is responsible for all the communication that MP has with other processes external to it.
Initiate-Departure	This process is responsible for monitoring the health of the neighbor and initiating agreement phase if the neighbor fails.
Initiate-Join	This process receives join requests from new members and acts on them in an appropriate manner.
Agreement-Process	This process receives and processes the agreement token.
Commit-Process	This process commits the removal or addition of a member from the group view.
TokenPool-Manager	This process manages the token pool . It sees to it that no tokens are lost and there are no duplicate tokens.
StatusTable-Manager	This process manages the status table and keeps track of the status of all members in the groupview.
GroupView-Manager	This process manages the group view and updates it when members leave and join.
Join-Initial	This process receives the initialization parameters and initializes the Status table, TokenPool and GroupView data structures in the new member.

39

```
Action-message
    char[10]            Action-type;
    Address             Member-Address;
```

Figure 4.3: Action-type message structure

TABLE 4.2: DIFFERENT ACTION ORIENTED DATA STRUCTURES.

Data Element	Action-type
Initiate-Agreement	initagree
Initiate-Commit	initcomit
Neighbor-Status	statmembr
Send-Init-Param	sendinitp
Update-GroupView	updtgview
Update-Status-Table	rmvemembr
Update-TokenPool	uptknpool

nition is given in Fig. 4.3. The different instances of occurrances of this data type with the action-type is given in Table 4.2.

GroupView-struct: This is defined in Fig. 4.4. The structure consists of View-number, Group-Size as strings and an array of Addresses. The size of this array is equal to the value specified as string in Group-size. *Initial-GroupView* and *Current-GroupView* are instances of this data type.

Neighbor: This data type is used to define various instances at which a neighbor address is required. It has two data elements, Initiator-Address and Neighbor-Address. The data is used to pass the address of the clockwise neighbor of the initiating address. It is defined in Fig. 4.5. *NeighborAddress, Current-Neighbor,* and *Neighbor-Member* are instances of this data type.

40

```
GroupView-struct
     char[5]                    View-Number;
     char[3]                    Group-Size;
     {
     Address                    Member-Address;
     } [atoi(Group-Size)]
```

Figure 4.4: GroupView message structure

```
Neighbor
     Address                    Initiator-address;
     Address                    Neighbor-address;
```

Figure 4.5: Neighbor-Address structure

StatusTable Structure: This is defined to specify the message structure for sending Status Table contents. It consists of a string of characters denoting the NumberOfEntries and an array of another structure consisting of member address and status as elements. The size of this array is given by the value specified in number of entries. It is defined in Fig. 4.6. *Initial-Status-Table* and *Current-Status-Table* are instances of this data type.

Token-data: This is defined to enunciate multiple instances of its occurrance. This structure consists of the elements initiator-address, member-address, and token-

```
StatusTable-struct
     char[3]                    NumberOfEntries;
     {
     Address                    Member-Address;
     char[10]                   Status-of-member;
     } [atoi(NumberOfEntries)]
```

Figure 4.6: StatusTable message structure

41

```
Token-struct
     Address              Initiator-address;
     Address              Member-Address;
     char[10]             Token-type;
```

Figure 4.7: Token structure

```
TokenPool-struct
     char[3]              NumberOfTokens;
     {
     Token-struct         Token-field;
     } [atoi(NumberOfTokens)]
```

Figure 4.8: TokenPool message structure

type. The token-type is used to distinguish between various tokens. The definition is given in Fig. 4.7. The explanation of various instances is given in data structure Token description.

TokenPool-struct: This data type consists of a character string denoting number of tokens and an array of token-field of type token-struct. The array size is given by the number of tokens field. The Fig. 4.8 gives the data definition. *Initial-TokenPool* and *Current-TokenPool* are instances of this data type.

Group View: This is a linked list used for storing the group view at each process site. The data structure consists of view number and group size as a character string and a linked list of member address and next member pointer. The tail of list is specified by null address in next-member field. The data structure is defined in Fig. 4.9.

Initial-Parameters-Receive: This is a character string formed by concatenating *Initial-Group View* , *Initial-Status-Table* and *Initial-TokenPool* into a single message.

```
GroupView
    char[5]                          View-Number;
    char[3]                          Group-size;
    Member-pointer{
    Address                          Member-Address;
    Member-pointer                   *Next-member;
    }
```

Figure 4.9: GroupView

Initial-Parameters-Send: This consists of the destination-address of the message and a message formed by concatenating *Current-GroupView*, *Current-TokenPool* and *Current-Status-Table*.

Join-Requests: This has requesting member address and the name of the group to join as components. The Request-member-address is of type address and Group-name is a character string.

Member-Status: The components of this data structure are member-address and a character string denoting the status of member as specified in status table.

Reset-timer: This is a message string "resetimer" to reset the watchdog timer used for taking periodic actions.

Status-message: This data type is defined to specify the data structure for querying and responding messages. The data definition is given in Fig. 4.10. *Status-Query-in* and *Status-Query-out* have the action field as "statquery". *Status-Report-in* and *Status-Report-out* have the action field as "statreprt".

Status-Table: This data is a linked list used for storing the status table at each process site. The data structure consists of number-of-entries in the list, and a linked list consisting of member address, member status and next member pointer. Null

43

```
Status-monitor message
    char[10]              action:
    Address               Initiator-Address:
    Address               Destination-Address:
```

Figure 4.10: Status-Monitoring message structure

```
Status-Table
    char[3]               NumberOfEntries:
    Status-pointer{
    Address               Member-Address;
    char[10]              Member-status:
    status-pointer        *Next-entry;
    }
```

Figure 4.11: Status-Table

address in next-member field denotes the tail of the list. The data structure is defined in Fig. 4.11. The different entries of member status are *DepartureA-greed, JoinAgreed, DeparturePending, DepartureAgreed* and *JoinPending.*

Timeout-message: This is a message "timeoutms" to denote that a time out has occurred.

Token: This is an instance of occurrance of Token-struct. Token-type is a string denoting type of token. There are 5 types of tokens and Table 4.3 gives a list of tokens and the Token-type corresponding to each of them.

TokenPool: This data is a linked list used for storing the tokenpool at each process site. The data structure consists of number of tokens as a character string and a linked list of member address. token-type and next member pointer. The tail of the list is specified by a null address in the next-member field. The data structure is defined in Fig. 4.12.

TABLE 4.3: DIFFERENT TOKENS WITH THEIR TOKEN-TYPES

Tokens	TokenType
Join Agreement Token	*Joinagree*
Failure Agreement Token	*Failagree*
Join Commit Token	*Joincomit*
Failure Commit Token	*Failcomit*
Join Requested Token	*Joinreqst*

```
TokenPool
    char[3]              NumberOfTokens;
    Token-pointer{
    Address             Member-Address;
    char[10]            Token-type;
    Token-pointer       *Next-token;
    }
```

Figure 4.12: TokenPool

TokenStatus: This is a data structure consisting of token-struct element specifying particular token and a character string giving the status of the token as old or new.

Update-Status: This is a data structure consisting of the address of the member specifying the member-address and a character string giving the new status of the member.

C. PROCESS SPECIFICATIONS

The individual processes are described in great detail in the following paragraphs. The function of each process, along with its inputs and outputs, is described. The shared data managed, if any is also specified. The algorithm used for implementing the function is also described.

1. FIFO-Channel-Layer

This process is responsible for all the communication with all the processes external to the MP executing in the member site. It receives *Status-query-out* from Initiate-Departure process and sends *Status-Query-out* to the counter-clockwise neighbor. It receives *Status-Report-in* from counter-clockwise neighbor and sends it to Initiate-departure process. On receiving a *Status-Query-in* from clockwise neighbor it sends it to Initiate-Departure Process. It receives *Status-report-out* from Initiate-Departure process and sends *Status-Report-out* to the site address specified. If the member site is the host of the group, the *Initialization-Parameters-send* is sent to the new member wanting to join the process group. If the member site is the new member wishing to join the group, this process receives *Initial-Parameters-receive* and initializes the storage elements in the MP protocol. Fig. 4.13 gives the interaction of FIFO-Channel-Layer process with other processes.

46
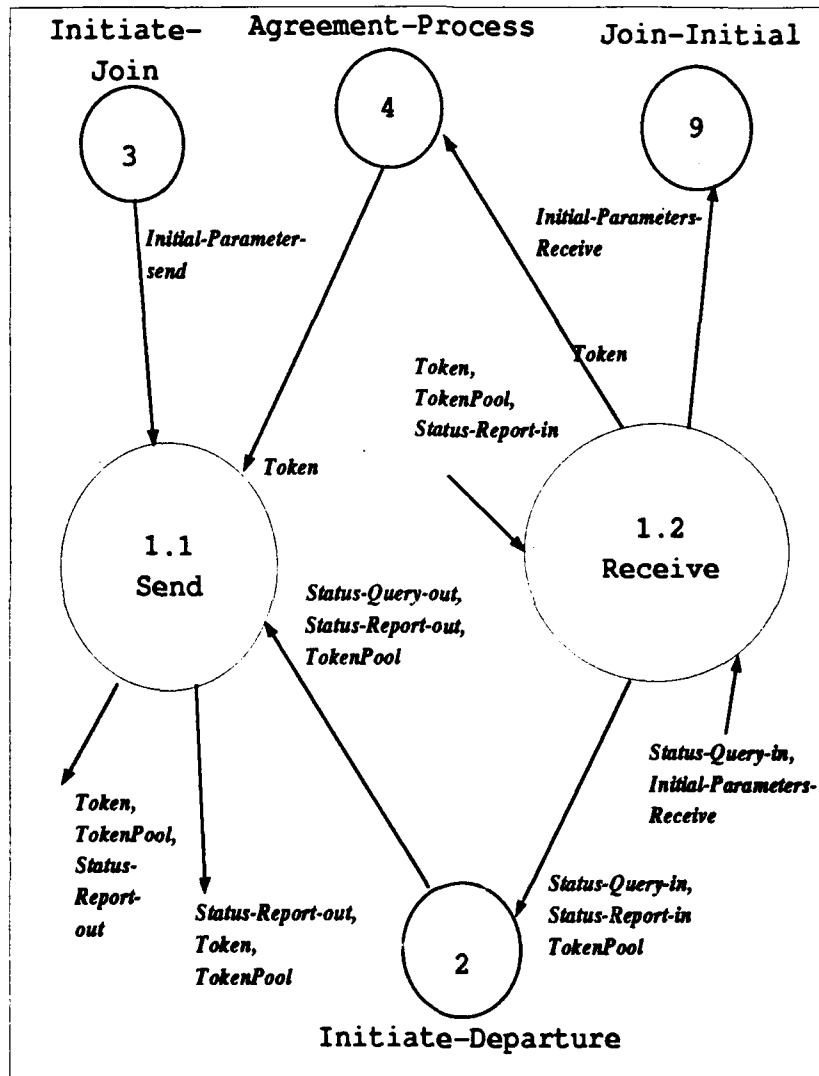
Figure 4.13: FIFO channel process

```
Send process
1   while (true)
2       Receive message from message queue;
3       Extract the destination from the message;
4       Send the message to destination specified;
5   end while;
end Send.
```

Figure 4.14: Send process

This process is subdivided into Send and Receive process. Send process has a message queue. All processes wishing to communicate with the processes external to MP, but executing locally, send messages to the message queue. The destination part of the message will specify the destination to be sent to. Receive process receives message streams from other members. The type of message sent by each member is embedded in the message. This process scans the type of message sent and sends the message stream to the appropriate process. Send process has 12 data flows.

Input data flows are *Status-Report-out, Token, TokenPool, Status-Query-out, Token and TokenPool*

Output data flows are *Status-Report-out, Token, TokenPool, Status-Query-out, Token and TokenPool*

The algorithm used for implementing the process is given in Fig. 4.14. Receive process has 9 data flows.

Input data flows are *Status-Report-in, Token, TokenPool, Status-Query-in and Initial-Parameters-receive*

Output data flows are *Status-Query-in, Status-Report-in, Token and Initial-Parameters-receive*

48

```
Receive process
1   while (true)
2         Receive message from other members;
3         Extract the type of message received;
4         Identify the destination process ;
5         Pass the message to the destination process;
6   end while;
end Receive.
```

Figure 4.15: Receive process

The algorithm for implementing the Receive function is given in Fig. 4.15.

2.   Initiate-Departure

This process checks for the health of its counter-clockwise neighbor by processing the Status report received from it. It keeps track of the time elapsed from the last query sent to the counter-clockwise neighbor. If the elapsed time is greater than a threshold it takes the following actions.

1. It initiates the agreement token for the process perceived to have failed.

2. It updates the address of the process to which query is to be sent based on *Group View* entries and *Status-Table* entries.

3. It updates the local *Status-Table* ..... *TokenPool.*

If it receives a query from a process other than its neighbor, it updates the status of neighbor and does the following actions.

1. It sends status report to the new querying process.

2. It sends the *Current-TokenPool* to the new querying process.

Fig. 4.16 gives the interaction of Initiate-Departure process with other processes. Initiate-Departure process is divided into 3 sub processes. They are
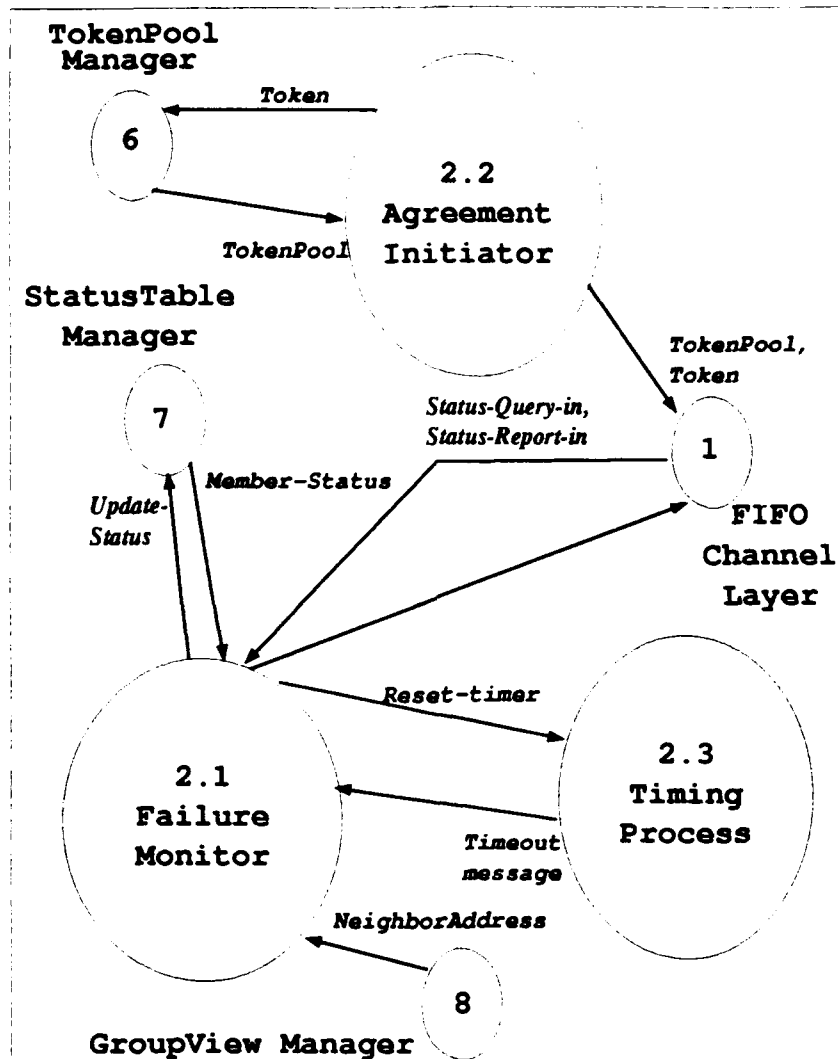
Figure 4.16: Initiate-Departure process

`Failure-Monitor`, `Agreement-Initiator`, and `Timing-process`. `Failure-Monitor` does the processing for failure detection. It receives status query from `FIFO-Channel-Layer` process and checks to see if the querying process address is the same as the previous address. If it is the same, it sends status report to the address specified. If it is different, it updates the new monitoring member as its clockwise neighbor. This process periodically queries the counter-clockwise neighbor and waits for a status report. If the status report is not received within a certain time, it shuts off communication from that process . It sends an *Initiate-Agreement* signal to `Agreement-Initiator` process. It then updates its neighbor from `GroupView-Manager` and sends query to the new process.

`Agreement-Initiator` process initiates an agreement process when the counter-clockwise neighbor is perceived to have failed. It receives a message from `Failure-Monitor` process when a process has failed. It indicates the address of the member perceived to have failed. It adds the agree token to the *TokenPool* and sends the token to the new member. `Timing-process` process keeps track of time for periodic actions. It signals if a timeout has occurred since the previous status report. It resets the timer at the receipt of the reset timer signal. `Failure-Monitor` process has 10 data flows.

Input data flows are *Status-Query-in, Status-Report-in , Member-Status, NeighborAddress, Timeout-message*

Output data flows are *Status-Query-out, Status-Report-out, Update-Status, Initiate-Agreement, Reset-timer*

The process specification is given in Fig. 4.17. `Agreement-Initiator` process has 6 data flows.

Input data flows are *TokenPool, InitiateAgreement*

```
Failure-Monitor process
1   while (true)
2        Read message from FIFO-Channel-Layer process;
         /* Formal algorithm is given in Figs. 2 and 3 of [Ref. ShDr] */
3        if (message == Status-Report-in)
4            wait till Timeout-message is received; /* line 11 */
5            reset timer
6            send Status-Query-out;
7        else if (message == Status-Query-in)
8            check Initiator-Address;
9            if (Initiator-Address == P_mon)
10               send Status-Report-out to Initiator-Address;
11           else
12               send Status-Report-out to Initiator-Address;
13               send TokenPool to Initiator-Address;
14               p_mon = Initiator-Address;
15           end if
16       else if (message == Timeout-message)
17           Agreement-Initiate process gets Initiate-Agreement message;
18           get new neighbor address from GroupView-Manager;
19           send Status-Query-out to new counter-clockwise neighbor;
20           reset timer;
21       end if
22  end while;
end Failure-Monitor.
```

Figure 4.17: Failure-Monitor

```
Agreement-Initiator process
1   while (true)
2       wait for Initiate-Agreement message;
3       read address of failed member;
4       read address of new neighbor;
        /* acwnbr(pᵢ) is failed member. */
        /* cwnbr(pᵢ) is new monitor. */
5       send agree_pᵢ (failed-member) to new neighbor;
6       send Token to TokenPool;
7       update status as DepartureAgreed;
8   end while;
end Agreement-Initiator.
```

Figure 4.18: Initiate-Agreement process

Output data flows are *TokenPool, Token, Token, Update-Status*

The process specification of this process is given in Fig. 4.18. The actions performed are given in a Pseudo-code form in lines 6-10 of Fig. 2. in the [Ref. ShDr].

Timing-process process has 2 data flows. Input data flow is *Reset-timer* and Output data flow is *Timeout-message*. The process specification is given in Fig. 4.19.

3. Initiate-Join

This process does all the steps involved in the process of adding a new member. If this process receives a *Join-Request* and the process is in the host, or if it receives *Send-Init-Param* message, then it perform· the following actions.

1. It receives the *Current-GroupView* from the GroupView-Manager.

2. It receives the *Current-StatusTable* from StatusTable-Manager.

3. It receives the *Current-TokenPool* from the TokenPool-Manager.

```
Timing-process process
1   while (true)
2       wait for timer-overflow or reset-timer message;
3       if timer-overflow
4           send Timeout-message;
5           disable timer;
6       else
7           wait for timer-overflow;
8           reset timer;
9           enable timer;
10          send timeout message;
11      end if;
12  end while;
end Timing-process.
```

Figure 4.19: Timing process

4. It sends all of the above information to the new member in a consolidated message.

If a *Join-Requests* is received and it is not in host member it sends a *Joinreqst* token to clockwise neighbor and adds the token to the *TokenPool.* This process has 8 data flows.

Input data flows are *Join-Requests, Current-TokenPool. Current-Status-Table, GroupView. Send-init-param*

Output data flows are *Initial-Parameters-Send. Token. Token*

The process specification is given in Fig. 4.20.

4. Agreement Process

This process does the agreement token processing. Whenever it receives a token it checks to see token type. If it is a commit token it sends a *Commit-Initiate* message to the Commit-Process. If it is an agreement token it does the following

```
Initiate-Join process
1   while (true)
2        wait for Join-Requests or Send-Init-Param;
3        if ((Join-Requests == true) and (host-process))
4            send Joinagree to clockwise neighbor;
5            update Status-Table and add Token to TokenPool;
6        else
7            send Joinreqst to clockwise neighbor;
8        end if;
9        if ((Send-Init-Parm == true)
10           get Current-TokenPool from TokenPool-Manager;
11           get Current-GroupView from GroupView-Manager;
12           get Current-StatusTable from StatusTable-Manager;
13           convert all these messages to Initial-Parameters-Send message.
14           send the message to the new member.
15       end if;
16   end while;
end Initiate-Join.
```

Figure 4.20: Initiate-Join process

actions.

1. Checks if it is a duplicate token.

2. Update *TokenPool* if it is not a duplicate token.

3. If token is a duplicate, it checks to see if the current process has to initiate commit, based on entries in the *Group View* and *Status-Table*.

If it is a join request token, it does the following actions.

1. Checks if it is the host process.

2. Sends the token to neighbor and add to *TokenPool* if it is not host process.

3. If it is the host, initialize the joining process by sending *Joinagree* token to the clockwise neighbor. Update *Status-Table* and add *Token* to *TokenPool*.

This process has 9 data flows.

Input data flows are *Token, Current-Neighbor, TokenStatus, Neighbor-Status*

Output data flows are *Token, Neighbor-member, Token, Initiate-Commit, Send-init-param*

The formal algorithm is given in Fig. 5 of [Ref. ShDr] . The process specification is given in Fig. 4.21.

5. Commit Process

The commit process is responsible for committing the removal or the joining of the member. This process waits for the initiate commit message specifying the address and the action to be performed. The functions performed by this process are

1. Receive *Initiate-Commit* from **Agreement-Process**.

```
Agreement-Process at p_i
1   while (true)
2       wait for Token;
3       if ( Token-type == (Joincomit or Failcomit))
4           send Initiate-Commit to Commit-Process;
5       else if ( Token-type == Joinreqst)
6           if (host-process)
7               send Joinagree to clockwise-neighbor;
8               update Status-Table and add Token to TokenPool;
9           else
10              send Joinreqst to clockwise-neighbor;
11              add Joinreqst to TokenPool;
12          endif;
13      else if ( Token-type == Joinagree or Failagree)
14          if ((Token - type ==Joinagree) and (Member - address is in GroupView))
14.1            exit;
15          if ( TokenStatus == old)
16              if ( Initiator-address == self-address)
16.1            ((status of Initiator-address == not operational )
16.2               && (status of all members between initiator and self is non operational))
17                  if more tokens present
18                      compare rank of all agreement tokens;
19                      if (rank of self is minimum) initiate commit;
20                      else update status as JoinPending or DeparturePending
21                      end if;
22                  else initiate commit for agreement process
23                  end if;
24              end if;
25          else
26              add Token to TokenPool and send Token to clockwise neighbor;
27              update status in Status-Table;
28          end if;
29      end if;
30  end while;
end Agreement-Process.
```

Figure 4.21: Agreement process

2. Check to see if it is a duplicate token.

3. If it is not a duplicate token then purge all commit tokens before the agreement token of the member being committed.

4. Update the status table entry by deleting entry corresponding to the member leaving.

5. Update *Group View*, and *ViewNumber* in the *Groupview* by deleting member if *Failcomit* and by adding member if *JoinComit*.

This process has 4 data flows.

Input data flows are *Initiate-Commit*

Output data flows are *Update-Group View, Update-Status-Table, Update-TokenPool*

The formal algorithm is given in Fig. 6 of [Ref. ShDr] . The process specification is given in Fig. 4.22.

6. TokenPool Manager

This Process manages the *TokenPool* which keeps a record of all tokens sent. It maintains and manages a linked list of token entities in a client-server relationship. ' The service is requested by the client process by writing a message onto a message queue. The service requested is embedded in the message. It does the following functions, depending on the service required by the client process.

1. It adds a *Token* to the tail of the list.

2. Removes all commit tokens occurring before a particular agreement token and adds the commit token to the tail of the list.

```
Commit-Process at $p_i$
1   while (true)
2        wait for Token;
3        if (TokenStatus == New);
4            send Update-GroupView message to GroupView-Manager;
5            send Update-Status-Table message to Status-Table-Manager;
6            send Update-TokenPool message to TokenPool-Manager;
7            send token to clockwise-neighbor process;
8            update host-process address and counter-clockwise neighbor address;
9            if ((host-process) and (Joincomit))
10               send Send-Init-Param message to Initiate-Join process;
11           endif;
12           if (DeparturePending or JoinPending)
13               if (rank of pending member is minimum)
14                   initiate-commit;
15               end if;
16           end if;
17       end if;
18  end while;
end Commit-Process.
```

Figure 4.22: Commit process

```
TokenPool-Manager process
1   while (true)
2       wait for message;
3       if (message-type == Tokenstat)
4           send status of token to client process;
5       else if (message-type == Tkpoolreq)
6           send contents of TokenPool to client process;
7       else if (message-type == Addtoken)
8           add token to the end of TokenPool ;
9       else if (message-type == inittknpl)
10          extract tokens from message and create TokenPool ;
11      else if (message-type == uptknpool)
12          remove all commit tokens before agreement token
13          remove Joinreqst token for member being committed
14          remove agreement token and add commit token in the TokenPool ;
14      end if;
15  end while;
end TokenPool-Manager.
```

Figure 4.23: TokenPool-Manager process

3. Initialises the *TokenPool* list based on the information received from the message.

4. Sends the content of the *TokenPool* to the requesting client.

5. Give the status of token as *New* or *Old*.

This process has 8 data flows.

Input data flows are *Token, 'oken, Token, Initial-TokenPool, Update-TokenPool*

Output data flows are *Current-TokenPool, TokenStatus, TokenPool*

The process specification is given in Fig. 4.23.

7. StatusTable Manager

StatusTable Manager keeps track of the status of the various members. The process acts as a server providing service to various clients. Depending on a client's request it does the following functions.

1. It sees if a member is listed in the *Status-Table*.

2. It gives the current status of the member in the *Status-Table*.

3. It updates the status of the member in the *Status-Table*.

4. It creates a message of all the members of the table with their current status for new members.

5. It creates a new *Status-Table* from the message received from host.

This process has 5 data flows.

Input data flows are *Update-Status-Table, Update-Status, Initial-Status-Table*

Output data flows are *Member-Status, Current-Status-Table*

The process specification is given in Fig. 4.24

8. GroupView Manager

GroupView-Manager manages the membership list and the view number. It interacts with other processes in a client server relationship and does the following functions depending on client's request.

1. Check if a given member is in the *GroupView*.

2. Add a new member to the end of the group.

3. Delete a given member from the group and maintain the logical ring.

```
StatusTable-Manager process
1    while (true)
2         wait for message;
3         if (message-type == Statmembr)
4             send status of member to client process;
5         else if (message-type == statblreq)
6             send contents of Status-Table to client process;
.7        else if (message-type == Updtstatus)
8             change status of member and add if not present already ;
9         else if (message-type == inittable)
10            extract Status-Table from message and create Status-Table ;
11        else if (message-type ==Removmem)
12            remove status-table entry of member from Status-Table;
13        end if;
14  end while;
end StatusTable-Manager.
```

Figure 4.24: StatusTable-Manager process

4. Send all the members of the current view to the joining member if the process is a host.

5. Create the *GroupView* from the message received from the host.

This process has 7 data flows.

Input data flows are *Neighbor-member, Initial-GroupView, Update-GroupView*

Output data flows are *GroupView, Current-Neighbor, Current-GroupView, NeighborAddress*

The process specification is given in Fig. 4.25.

9.   Join Initial

This process does all the initialization when the process joins a new group.

It receives *Initial-parameters-receive* from FIFO-channel-Layer process and extracts

```
GroupView-Manager process
1   while (true)
2        wait for message;
3        if (message-type == Updtgview)
4            add or remove member and increment View.Number.
5        else if (message-type ==gpviewreq)
6            send contents of group view to client process;
7        else if (message-type ==neibraddr)
8            send the address of clockwise neighbor of member;
9        else if (message-type == initgview)
10           extract GroupView from message and create GroupView ;
11       end if;
12  end while;
end GroupView-Manager.
```

Figure 4.25: GroupView-Manager process

Initial-TokenPool, Initial-Status-Table and Initial-GroupView from the message. It sends *Initial-TokenPool* as a message to TokenPool-Manager. It sends *Initial-GroupView* as a message to GroupView-Manager. It sends *Initial-Status-Table* to StatusTable-Manager.

D.   IMPLEMENTATION ON UNIX MACHINES

In this section, some of the communication protocols that can be used for creating a ring of First-In-First-Out communication channels are discussed. A ring of FIFO channels is created by logically ordering the group members in a ring and interconnecting them through FIFO communication channels. The relative merits and problems of various inter process communication (IPC) protocols available in UNIX are studied . Since the implementation of a ring of FIFO communication channels requires interaction of two or more processes, study of various methods for communicating between processes is necessary. In UNIX, there are different methods for communication and we discuss each one of them and see how these protocols can be used in implementing the algorithm. based on a logical ring of members in a group.

63

This sort of communication is not limited to only communication between two systems but also processes on a single system. We deal with the following types of IPC's for intra-machine communication.

- Pipes

- FIFO's (named pipes)

- Message Queues

We will be dealing with the following IPC's for inter-host communication

- Sockets

- Transport Layer Interfaces (TLI).

We deal with all IPC techniques for setting up a client-server relationship and deal with all types of IPC's for the same host and discuss their relative merits and problems.

1. Pipes

Pipes [Ref. SR90] [Ref. Roch] [Ref. CM89] are provided by all flavors of UNIX. A pipe provides a one-way flow of data. A pipe is created by the pipe system call.

int pipe(int *filedes);

Two file descriptors are returned- *filedes[0]* which is open for reading and *filedes[1]* which is open for writing. Pipes are of little use within a single process. Pipes are typically used to communicate between two different process in the following way. First, a process creates a pipe and then forks to create a copy of itself. Next the parent process closes the read end of the pipe and the child closes the write end of the pipe. This provides a one way flow of data between processes. For a two way flow

of data, two pipes are to be created and one is used for each direction. The actual steps are given below.[Ref. SR90]

- Create pipe1, create pipe2,

- fork,

- parent closes read end of pipe1,

- parent closes write end of pipe2

- child closes write end of pipe1,

- child closes read end of pipe2.

The biggest disadvantage with pipes is that they can only be used between processes that have a parent process in common. This is because a pipe is passed from one process to another through the **Fork** system call and the fact that all open files are shared by the parent and the child after a *Fork* . There is no way for two totally unrelated processes to create a pipe between them and use it for IPC.

2. FIFOs

FIFO [Ref. SR90, Roch, CM89] stands for *First In, First Out* . A Unix FIFO is similar to a pipe. It is a one way flow of data with the first byte written to it being the first byte read from it. Unlike the pipes FIFOs have a name attached to it, allowing unrelated processes to access a single FIFO. FIFO is created by the **mknod** system call.

int mknod(char *pathname, int *mode*, int *dev*);

The *pathname* is a normal Unix pathname and this is the name of FIFO. The *mode* argument specifies the file mode access mode for the file (read, write permissions for owner, group, world). The *dev* argument is ignored for a FIFO. Once the FIFO is

65

created it must be opened for reading and writing using the *open* system call. Three system calls are required for creating and opening FIFOs for reading and writing . The sequence of actions involved are

- Create FIFO

- Open FIFO for reading and get file descriptor for reading

- Open FIFO for writing and get file descriptor for writing

Only one command does the same thing for pipes. One of the rules followed by pipes or FIFOs is that *write* is guaranteed to be atomic if the write is less than the capacity of a pipe or FIFO. The capacity is greater than 4 kbytes. If it is greater then there is a possibility of data and atomicity is not guaranteed. There is some care to be taken in the order of open calls to avoid a deadlock condition. When the client opens FIFO1 for writing, it waits for the server program to open FIFO1 for reading. If the first call of server is for FIFO2 instead of FIFO1. each process would be waiting for the other, and neither would proceed. This leads to a deadlock. One of the disadvantages of pipes and FIFOs are that they are stream I/O models. The message boundaries are delineated with the newline character and it is not possible to have structured messages.

3. Message Queues

Message Queues [Ref. SR90, Roch] are used to pass messages between processes in System V implementation. Processes read and write to arbitrary queues. There is no requirement that any process be waiting to read before some other process is allowed to write a message to that queue. This is unlike the case of pipes and FIFOs. It is possible for a process to write a message in the queue and exit and have another process read the messages at a later time. Each message on a queue has the following attributes:

66

```
Message buffer structure
    struct        msgbuf {
    long          mtype;        * message type is greater than zero *
    char[]        mtext;        * message data *
    }
```

Figure 4.26: Message queue structure

- long integer *type*;

- *length* of the data portion of the message.

- *data*(if the length is greater than zero).

The message queue can be thought of as a linked list of messages. A new message queue is created or the old one accessed using the *msgget* system call. The value returned by msgget is the queue identifier *msqid*. Once a message queue is opened, we put the message in the queue using the *msgsnd* system call.

int msgsnd(int *msqid*, struct msgbuf **ptr*, int *length*, int *flags*);

The ptr argument points to a structure with the following template. Fig. 4.26 specifies the structure of a message in a message queue. Message type must be greater than zero since it is used by *msgrcv* as a special indicator to get messages of that particular type only. This is very useful in multiplexing messages. One way of multiplexing a single server with multiple clients, is to have one message type for communication from clients to server and to embedd in the message, the type of message the client will respond to. For example, the message type for client to server could be 1 and the client will include their process id in the message. The server will use this process id and use it as the type when sending messages to that client. The client will receive only messages specified by its process id by specifying its process id as the type of message it wants to receive. We now analyze a program where a server gets messages

from 4 clients on a message queue and processes their requests and replies to them over the queue.

In this program the server maintains a linked list of the members of a Groupview. There are four clients who can do the following functions.

- Initialise the Membership list.

- Add a member to the list.

- Remove a member from the list.

- Request the list of members in the current list.

The server gets this information from these clients and uses their process-ids to send the message back to them. The server waits for messages from clients and acts on them as they arrive. When there is more than one message they are acted on the order of arrival. The server will always be waiting in the *msgrcv* system call. All the IPCs discussed till now deal with communication only within the same host. Now we deal with the methods of communicating over different hosts.

4. Sockets

Sockets [Ref. SR90, CM89] are basically used for Network I/O as opposed to file I/O in the same machine. This needs more details and options. For example the details and options that would be necessary are given briefly in a few sentences. Typical client-server relationship is not symetrical, i.e. the actions to be performed by a client are different from the actions to be performed by servers. To initiate a connection request, the program must know which role it is to play. The network connection could be connection oriented or connectionless and each has a different sequence of actions to perform. The names are important in networking because

68

verification of authority for requested services should be possible. For network protocols, message boundaries have a lot of significance. We deal mainly with connection oriented networks since we are interested in a FIFO channel which is not guaranteed by connectionless protocols. The transport is based on TCP protocols.

To do Network I/O, the first thing a process should do is to call the *socket* system call specifying the type of communication protocol required. The socket could be

- stream socket,(connection oriented protocol)

- datagram socket, (connectionless service)

- raw socket,

- sequenced packet socket.(more than one message sent with sequence numbers)

This call returns an integer similar to a file descriptor called *sockfd*. The *bind* system call binds the local address and local process for a connection oriented server. *listen()* and *accept()* system calls are used for foreign address and foreign process in a connection oriented server. *connect()* system call is used by connection oriented clients. The client knows the socket address by binding of the address by the server. The server address is known to the client and the client knows the port number that the server uses for socket connection.

5. Transport Layer Interface

Transport Layer Interface(TLI) [Ref. SR90, SUN] provide an interface to the transport layer of the OSI model. It is a set of library functions that hide the actual streams interface to the networking system. Two processes that are communicating are called transport endpoints in TLI. The transport provider is a set of routines in

the host computer that provide communication support to the user process. Some of the elementary functions in TLI are

- *t-open* which is used to establish a transport endpoint by specifying the particular transport provider.

- *t-bind* assigns an address to the transport endpoint.

- *t-alloc* allocates space for various data structures used in all the TLI functions.

- *t-connect* is used for connecting a client to a server in a connection oriented network.

- *t-listen* is called by servers waiting for requests from clients.

- *t-accept* is called to accept connection indicated by *t-connect* function.

- *t-snd and t-rcv* functions are used to send and recieve data.

V. AN EXAMPLE

In this chapter, we give an example of the execution of this protocol. The example starts with a fixed number of members and simulates a sequence of failures and joins to the membership. We then analyze the group view at all member sites to see if they are identical for all view numbers.

A. INITIAL CONDITIONS

Assume six members in a group p_0, p_1, p_2, p_3, p_4, and p_5 who form a logical ring $\mathbf{p_0} \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_5$. All members have this structure in their groupview. Assume a *ViewNumber* of 6 at all places. The *StatusTable* has no entries in it corresponding to any member in the group. The *TokenPool* at all sites has the $commit_{\mathbf{p_0}}(p_5)$ token in the list. All members recognize member $\mathbf{p_0}$ as the host of the group. In this example, p_1 is the clockwise neighbour of p_0, p_2 is the clockwise neighbour of p_1 and so on. The monitoring action consists of the member querying its counter-clockwise member and getting a status report from it. Thus p_1 queries p_5 and gets a status report from it. Likewise (p_1, p_0), (p_2, p_1), (p_3,p_2), (p_4,p_3), (p_5,p_4) form querying and reporting pair. Fig.3.1 gives the members and other related information. The sequence of joins and failures to be checked for are

1. Failure of member p_2,

2. Join request of p_6 arriving at p_1 after agreeement phase for p_2 is over,

3. Failure of member p_5 ,

4. Join request of member p_7 almost immediately sent to member p_0, and

71

5. Failure of member p_0.

The next section describes in detail the sequence of actions taking place at all member sites for this sequence of events and gives a snapshot of *GroupView*, *StatusTable* and *TokenPool* at different points of time.

B.   EXPLANATION OF THE EXAMPLE

The example used here consists of three parts. The first part deals with a single failure while the second part deals with a single join. The third part deals with multiple failures and joins.

1.   Failure of a Single Member

The snapshot of parameters at all members is given in Table 5.1. The failure of p_2 is detected by p_3 when the timer in p_3 timeouts before it can receive a status report from p_2. When this happens p_3 shuts off communication from p_2 and starts the agreement phase of the MP to agree on the failure of p_2. The `Failure-Monitor` process at p_3 queries its local groupview manager and finds that the new member that it has to monitor is p_1. It then sends a status query to p_1. It also sends an agreement token $agree_{p_3}(p_2)$ for the failure of p_2 to p_4. It updates the *TokenPool* with this agreement token and a status table entry for p_2 is created with entry as *Failagree*.

When p_1 receives the status query from p_3 it compares the sender address with its previous monitor address and finds it to be different. It shuts off communication to p_2 and makes p_3 its clockwise neighbor. p_1 also sends its *Tokenpool* and status report to p_3. The *TokenPool* consisits of only the commit token for join of p_5. This is because we have assumed that the last change to membership view was the join of p_5. The commit tokens are garbage collected when another commit token is received and the commit tokens ,occuring before the agreement token for the same

TABLE 5.1: SNAPSHOT OF INITIAL CONDITION

Event: This is the snap shot of the member parameters initially				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_0, p_1, p_2, p_3, p_4, p_5	$\{p_0$, p_1, p_2, p_3, p_4, $p_5\}$, 6	$commit_{p_0}(p_5)$	Nil	

member, are removed. Since join of p_5 was the last change, $commit_{p_0}(p_5)$ is in the *TokenPool* till there is a new change. The status of the token is checked from the *Tokenpool*. This token is seen to be a duplicate by p_3 and it takes no action on it.

The Agreement token $agree_{p_3}(p_2)$ goes from p_3 to p_4 to p_5 to p_1 to p_3. The snapshot of parameters at all members is given in Table 5.2. The *TokenPool* at all these member sites have the agreement token as the last token and the status table entry at all sites at this point have the entry corresponding to p_2 as *Failagree*. When the token reaches p_3 it is seen as a duplicate token. p_3 then sees that it is the initiator and initiates the Commit phase. p_3 sends commit token to its *TokenPool* and removes the entry corresponding to p_2 in Status Table. It also removes the token $commit_{p_0}(p_5)$ which occurs before $agree_{p_3}(p_2)$, as per the garbage collection protocol. It also removes $agree_{p_3}(p_2)$ from *TokenPool* and $commit_{p_3}(p_2)$ is the only token in the *TokenPool*. The View number is incremented to 7 and p_2 is removed from groupview. The snapshot of parameters at all members is given in Table 5.3. It sends the commit token to p_4. These events happen at all the member sites and they update their *GroupView* and *StatusTable* accordingly. When p_3 gets the commit token back from p_1, it sees it to be a duplicate token and hence does not take any action, thus completing the commit phase at all member sites. The snapshot at all member sites is given in Table 5.4.

TABLE 5.2: SNAPSHOT AT THE END OF AGREEMENT PHASE

Event: The snapshot at the end of agreement phase				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_0, p_1, p_4, p_5	$\{p_0, p_1, p_2, p_3, p_4, p_5\}$, 6	$commit_{p_0}(p_5)$ $agree_{p_3}(p_2)$	p_2 Failagree	
p_3	same as above	same as above	same as above	Initiates the Commit phase of p_2

TABLE 5.3: SNAPSHOT WHEN ONLY p_3 HAS COMMITTED

Event: The snapshot when only p_3 only has committed				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_3	$\{p_0, p_1, p_3, p_4, p_5\}$, 7	$commit_{p_3}(p_2)$	Nil	p_3 commits to p_2's departure
p_0, p_1, p_4, p_5	$\{p_0, p_1, p_2, p_3, p_4, p_5\}$, 6	$commit_{p_0}(p_5)$ $agree_{p_3}(p_2)$	p_2 Failagree	

TABLE 5.4: SNAPSHOT WHEN ALL MEMBERS HAVE COMMITTED p_2

Event: This is the snapshot when all members commit p_2				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_0, p_1, p_3, p_4, p_5	$\{p_0, p_1, p_3, p_4, p_5\}$, 7	$commit_{p_3}(p_2)$	Nil	

2. Join of a Single Member

When the *JoinRequest* of a potential member p_6 arrives at p_1, the member sees that it is not the host and creates a *StatusTable* entry for p_6 as *Joinreqst* and also updates its *Tokenpool* with $joinreq_{p_1}(p_6)$. It sends this token to its clockwise neighbor p_3. This token goes to p_4 and p_5 before it reaches the host p_0. The snapshot at this point is given in Table 5.5. When the host gets this token it initiates the join agreement phase by circulating the agreement token $agree_{p_0}(p_6)$. This token, as it traverses round the ring, is added to the *TokenPool* of all the members.

When the host receives the agreement token again it initiates the commit phase. It updates its *GroupView*, *Tokenpool* and *StatusTable* and sends $commit_{p_0}(p_6)$. It then sends to p_6 the current contents of its *GroupView*, *StatusTable* and *TokenPool* and makes p_6 its counter-clockwise neighbor. p_6, on receiving the initialisation parameters, commits itself to the group view . The snapshot at this point of time is given in Table 5.6. p_6 computes p_5 as *its counter-clockwise neighbor* and sends a status query to it. p_5, on receiving the status query from p_6, makes it the clockwise neighbor and sends the *TokenPool* to it. The *TokenPool* contains $agree_{p_0}(p_6)$ which is not a duplicate token in p_6. This token is ignored by p_6 because it sees that it is the agreement token for join of a member already in the groupview. The commit token goes to all the members and when it receives a commit token, the $Joinreq_{p_1}(p_6)$ is removed from the *TokenPool*. The snapshot at the end of the commit phase is given in Table 5.7.

3. Multiple Failures and Joins

In this subsection, an example where a member joins the group and the initiator of the agreement phase fails after passing the token is given. Another member fails at the same time. The join request of potential member p_7 is received by the host p_0 and it initiates the agreement phase. It sends the agreement token $agree_{p_0}(p_7)$ to

TABLE 5.5: SNAPSHOT BEFORE INITIATING AGREEMENT FOR A JOIN

Event: This is the snapshot before agreement phase for p_6				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_0	$\{p_0, p_1, p_3, p_4, p_5\}, 7$	$commit_{p_3}(p_2)$	Nil	When p_0 receives $Joinreq_{p_1}(p_6)$, it initiates the agreement phase for join
p_1, p_3, p_4, p_5	$\{p_0, p_1, p_3, p_4, p_5\}, 7$	$commit_{p_3}(p_2)$ $joinreq_{p_1}(p_6)$	p_6 $Joinreqst$	

p_1 and fails. p_1 perceives the failure of the host and initiates the agreement phase for its failure. It updates its counter-clockwise neighbor and sends a query to p_6. p_6 updates p_1 as its clockwise neighbor and sends its report and $TokenPool$ to p_1. p_1 sends $agree_{p_0}(p_7)$ followed by $agree_{p_1}(p_0)$ to p_2.

If p_0 had failed without initiating the agreement phase, p_7 would have waited for a time period and would have sent the join request again. By that time the failure of p_0 would be committed, p_1 would be the host, and would initiate the agreement phase for the joining of p_7.

At this point, p_5 is perceived to have failed by p_6 and p_6 initiates the agreement phase. The token is passed around the ring and it reaches p_1 before agreement tokens $agree_{p_0}(p_7)$ and $agree_{p_1}(p_0)$ circulate back to p_1. When $agree_{p_0}(p_7)$ reaches p_1, it finds that it is a duplicate token. It also finds that other agreement tokens are also present and that the initiator of agreement phase has failed and,

TABLE 5.6: SNAPSHOT WHEN p_6 IS COMMITTED AT THE HOST p_0

Event: This is the snapshot when p_6 is committed by p_0				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_0	$\{p_0, p_1, p_3, p_4, p_5, p_6\}$, 8	$commit_{p_0}(p_6)$	Nil	
p_6	same as above	same as above	same as above	p_6 commits itself and begins monitoring p_5 although it is not yet committed by p_5.
p_1, p_3, p_4, p_5	$\{p_0, p_1, p_3, p_4, p_5\}$, 7	$commit_{p_3}(p_2)$ $joinreq_{p_1}(p_6)$ $agree_{p_0}(p_6)$	p_6 $Joinagree$	

TABLE 5.7: SNAPSHOT WHEN ALL MEMBERS HAVE COMMITTED p_6

Event: This is the snapshot when all members commit p_6				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
$p_0, p_1, p_3, p_4, p_5, p_6$	$\{p_0, p_1, p_3, p_4, p_5, p_6\}$, 8	$commit_{p_0}(p_6)$	Nil	

therfore, it must initiate the commit phase. It then computes the rank of all the members whose agreement token is pending and it finds that there are agreement tokens with smaller ranks. For example the rank of p_0 is 0, p_5 has a rank of 4, and p_7 has a rank of 6. So, it updates the status of p_7 as *Joinpending*. It is assumed that $agree_{p6}(p_5)$ is at p_4 and has not reached p_6. Table 5.8 gives the snapshot at this time. The snapshot gives only the parameters of the operational members in the current *GroupView*.

p_1 then receives the fail agreement token for p_0 , finds its rank as minimum, and initiates the commit for p_0. p_6, on receiving $agreep_6(p_5)$, finds that the rank of p_5 is not minimum and updates the status table entry for p_5 as *Failpending*. When $commit_{p_1}(p_0)$ is processed at p_6, it inspects other agreement tokens and finds that p_5 has the minimal rank. It initiates the commit phase for p_5. When the commit for p_5 is processed at p_1, it initiates the commit for join of p_7. The snapshot is given in Table 5.9. It should be noted that, in this snapshot, p_3 has not yet received the commit tokens for both p_5 and p_7. At the end of these commit actions all the members will have identical groupviews. However, the views committed at different members may be different at a particular instant. From this example, it is seen that the MP is robust for multiple failures and joins occuring at almost the same time.

TABLE 5.8: SNAPSHOT SHOWING MULTIPLE AGREE TOKENS

Event: Snap shot to show pending state at the time of commit				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
p_1	$\{p_0,\ p_1,\ p_3, p_4, p_5, p_6\}$, 8	$commit_{p_0}(p_6)$ $agree_{p_0}(p_7)$ $agree_{p_1}(p_0)$ $agree_{p_6}(p_5)$	$p_7\ JoinPending$ $p_0\ Failagree$ $p_5\ Failagree$	Initiates commit for failure of p_0 and suspends $commit_{p_1}(p_7)$.
$p_3,\ p_4$	$\{p_0,\ p_1,\ p_3, p_4, p_5, p_6\}$, 8	$commit_{p_0}(p_6)$ $agree_{p_0}(p_7)$ $agree_{p_1}(p_0)$ $agree_{p_6}(p_5)$	$p_7\ Joinagree$ $p_0\ Failagree$ $p_5\ Failagree$	
p_6	$\{p_0,\ p_1,\ p_3, p_4, p_5, p_6\}$, 8	$commit_{p_0}(p_6)$ $agree_{p_6}(p_5)$ $agree_{p_0}(p_7)$ $agree_{p_1}(p_0)$	$p_5\ Failagree$ $p_7\ Joinagree$ $p_0\ Failagree$	sets status of p_5 as $failpending$ and takes no further action on receiving $agree_{p_6}(p_5)$

79

TABLE 5.9: GROUPVIEW FOR SUCCESSIVE VIEW NUMBERS

Event: Snapshot at commit of p_7 at p_1				
Member(s)	GroupView, ViewNumber	TokenPool	StatusTable	Action
$\mathbf{p_1}$	$\{\mathbf{p_1},\ p_3,\ p_4,\ p_6,\ p_7\}$, 11	$commit_{\mathbf{p_1}}(p_0)$ $commit_{\mathbf{p_6}}(p_5)$ $commit_{\mathbf{p_1}}(p_7)$	Nil	$commit_{\mathbf{p_1}}(p_7)$ is sent
p_7	same as above	same as above	same as above	
p_3	$\{\mathbf{p_1},\ p_3,\ p_4,\ p_5,\ p_6\}$, 9	$agree_{\mathbf{p_0}}(p_7)$ $agree_{\mathbf{p_6}}(p_5)$ $commit_{\mathbf{p_1}}(p_0)$	$p_7\ Failagree$ $p_5\ Failagree$	
p_4	same as above	same as above	same as above	
p_6	$\{\mathbf{p_1},\ p_3,\ p_4,\ p_6\}$, 10	$agree_{\mathbf{p_0}}(p_7)$ $commit_{\mathbf{p_1}}(p_0)$ $commit_{\mathbf{p_6}}(p_5)$	$p_7\ Failagree$	

80

VI. CONCLUSIONS AND FUTURE DIRECTIONS

A. CONCLUSIONS

In this thesis, a decentralized mechanism for providing a consistent group view has been presented. This approach is different from other approaches, most of which are centralized in nature for providing a consistent group view. The proposed approach is efficient in that it requires only 2n messages for committing a change to the membership if the group contains n members. The number of messages is the same for the failure of any member. The protocol is being implemented on a network of SUN workstations. The different types of system calls for implementing the protocol have been identified and tested. The programs for various *client-server* communication patterns used for interfacing various functions have been developed and tested.

B. FUTURE WORK

There is a considerable amount of work that can be done as a continuation of this thesis. The coding of the protocol needs to be completed and its functioning observed. Various experiments should be run to characterize the latency of committing membership changes and compared with the centralized protocol implemented in the same environment. Ideally, experiments should measure the performance seen for reliable multicast primitives. Another extension to this thesis is to propose a formal proof for the correctness of the protocol.

APPENDIX A

A. GROUPVIEW SERVER

```
#include    <stdio.h>
#include    "gvmesg.h"
#include    "msgq.h"

/*
 * This is a server to maintain and update the group view. It receives
 * requests from other client processes and acts according to the
 * clients request.
 */

GVMesg          group_view_mesg;

typedef    struct list_node *MembPtr;

typedef struct list_node {
    char      memb_addr[9];
    MembPtr      next;
} MembListNode;

struct view {/* the group view structure*/
    int view_number;
    int group_size;
} group_view;
    MembPtr    tempptr, headptr, tailptr,tempptr1;


main()
{
    int    id, prid;
    long    key;
    /*
     * create message queue if required.
     */
    key =1;
    prid= getpid();

    if ( ( id= msgget(GVSER,( PERMS | IPC_CREAT))) < 0 ) {
        err_sys("server: cant get message queue for GroupView server");
        }
    /*
     * do it eternally as an iterative server
     */
    while (key == 1)     {
```

```
            server(id);
      }

      exit(0);
}

server(id)
int      id;
{
      int      loop_index, loop_index1, loop_index2, loop_index3, grpsize;
      int      grpview_num, num_bytes_read, m1, m2;
      long     proc_id_long;
      char     errmesg[256], *sys_err_str();

      /*
       * Read the message from the message queue
       */

        /* type for client to server messages  */
      group_view_mesg.mesg_type = 1L;

      if ( ( num_bytes_read= gview_mesg_recv(id,&group_view_mesg)) <= 0)
          err_sys("server: message read  error");

      /*
       * Convert the process id embedded as a long integer for sending
       * the reply back to the same client.
       */

      proc_id_long =atol(group_view_mesg.proc_id);

      group_view_mesg.mesg_type = proc_id_long;


      /*
       * check for the message header part. there are seven possibilites
       * if the header says "Uprmgview" then the member is removed
       * from group view. If the header says "Upadgview"then add the
       * member to the end of the group. If the header is "initgview"
       * the initial group view list is created from the contents of the
       * message. If the header is "neibraddr" the anti-clockwise
       * neighbor address is sent back to client. If header is "rankmembr"
       * the rank of the members specified is given. If the header is
       * "hostaddrs" the address of the host is given to the client.
       * If the header is "gpviewreq" the contents of the group view list
       * is sent as a message.
       */

      if  (strcmp(group_view_mesg.msg_header, "Uprmgview") == 0)  {

          /*
```

```
 * read in the members from the list and compare
 * with the member to be removed. if a match
 * is found then remove the member .
 * change the link address so that the link is not broken
 */

tempptr = headptr;

/*
 * extract the member address portion from the list and
 * compare with the member to be removed. If there is no match
 * go to the next member till the complete list is covered.
 */

for ( loop_index = 0; loop_index < group_view.group_size;
loop_index++)  {

    if (strcmp( tempptr->memb_addr,
    group_view_mesg.mesg_data[0].member_address) != 0) {

        tempptr1 = tempptr;
        tempptr = tempptr1->next;
    }

/*
 * If the match is for the host( first member) remove host
 * and update the pointer to the list
 */

    else if (loop_index == 0) {
        tempptr = headptr;
        headptr = tempptr->next;
        free(tempptr);
        break;
    }

    else {

/* If a match is found in the middle of the list change
 * the link to reform the list.
 */

        tempptr1->next = tempptr->next;
        free(tempptr);
        if (tempptr1->next == NULL)
          tailptr = tempptr1;
        break;
    }
}

/*
```

```
    * read in the current values of view number
    * and number of members
    * decrease the number of members and increase
    * the view number.
    */

    group_view.view_number++;
    group_view.group_size--;


}

else if (strcmp(group_view_mesg.msg_header, "Upadgview") == 0)  {

    /*
     * This segment of program  creates storage for new member
     * it updates the view number and adds the address of
     * the new member to the list.
     *
     */

    tempptr = (MembPtr) malloc( sizeof( MembListNode));

    /*
     * add the new member to the end of
     * membership file. The first member is always
     * the host and the succesive entries in the file
     * denote the ring structure of group membership
     */

    strcpy(tempptr->memb_addr,
    group_view_mesg.mesg_data[0].member_address);
    tempptr->next = NULL;
    tailptr->next = tempptr;
    tailptr = tempptr;

    /*
     * read in the current values of view number
     * and number of members
     * increment the view number and the number
     * of members
     */


    group_view.view_number++ ;
    group_view.group_size++   ;


}
else if (strcmp(group_view_mesg.msg_header, "init_view") == 0)  {
```

```c
/*
 * this part of the program initialises the
 * list structure and generates the initial group view.
 */

group_view.view_number = atoi(group_view_mesg.view_num);
group_view.group_size  = atoi(group_view_mesg.num_mem);

/*
 * create the list structure. make the
 * pointer to the first element the header
 */

for (loop_index1 = 0; loop_index1 < group_view.group_size;
 loop_index1++)  {

    tempptr = (MembPtr) malloc( sizeof( MembListNode));
    if (loop_index1 ==0) {
        headptr = tempptr; /* head of the list */
        tailptr = headptr; /* initial tail  */
    }

    /*
     * copy the number of members iteratively.
     */

    strcpy(tempptr->memb_addr,
    group_view_mesg.mesg_data[loop_index1].member_address);
    tempptr->next = NULL;
    tailptr->next = tempptr;
    tailptr = tempptr;
}
}
else if (strcmp(group_view_mesg.msg_header ,"gpviewreq")== 0 ){

grpview_num = group_view.view_number;
grpsize = group_view.group_size;

/*
 * integer to ascii conversion routine for
 * view number. the string is a null terminated
 */

itoa(grpview_num, group_view_mesg.view_num, 5);

/*
 * integer to ascii conversion routine for number
 * of members .the string is a null terminated
 */
```

```c
        itoa(grpview_num, group_view_mesg.num_mem, 3);
        printf("group size is %d\n", grpsize);
        tempptr = headptr;

        /*
         * read the list completely and create the message to be
         * sent back. The process index is used for sending the
         * message back to the client.
         */

        for (loop_index2 = 0; loop_index2 < grpsize;
        loop_index2++) {
            strcpy(
            group_view_mesg.mesg_data[loop_index2].member_address ,
            tempptr->memb_addr);
            group_view_mesg.mesg_data[loop_index2].
            member_address[8] = NULL;
            tempptr1 = tempptr;
            tempptr = tempptr1->next;
        }

        group_view_mesg.mesg_type = atol(group_view_mesg.proc_id);
        group_view_mesg.mesg_len = grpsize*12 + 22;
        printf("message length =%d\n",group_view_mesg.mesg_len);
        gview_mesg_send(id, &group_view_mesg);
}
else if (strcmp(group_view_mesg.msg_header, "hostaddrs") == 0) {

        /*
         * send the address pointed by the headptr
         * as the host of the group.
         */

        tempptr = headptr;
        strcpy(group_view_mesg.mesg_data[0].
        member_address, headptr->memb_addr);
        group_view_mesg.mesg_type = atol(group_view_mesg.proc_id);
        group_view_mesg.mesg_len = 12 + 22;
        gview_mesg_send(id, &group_view_mesg);

}


else if (strcmp(group_view_mesg.msg_header, "neibraddr") == 0) {

        /*
         * read in the members from the list and compare
         * with the member address given. if a match
         * is found then identify the member previous to
         * the match as anticlockwise neighbor
         */
```

```c
tempptr = headptr;

/*
 * extract the member address portion from the list and
 * compare with the member address given. If there is no
 * match continue  till the complete list is covered.
 */

for ( loop_index = 0; loop_index < group_view.group_size;
loop_index++)  {

    if (strcmp( tempptr->memb_addr,
    group_view_mesg.mesg_data[0].member_address) != 0) {

        tempptr1 = tempptr;
        tempptr = tempptr1->next;
    }

/*
 * If the match is for the host( first member)
 * tailptr( last member)is the anticlockwise neighbor
 */

    else if (loop_index == 0) {
    strcpy(group_view_mesg.mesg_data[0].
    member_address, tailptr->memb_addr);
        break;
    }

    else {

/* If a match is found in the middle of the list
 * anticlockwiise neighbor is the previous member.
 */
        strcpy(group_view_mesg.mesg_data[0].
        member_address, tempptr1->memb_addr);

        break;
    }
}

/*
 * Send the message back to the client. fill the
 * message structure with all relavant details.
 */

group_view_mesg.mesg_type = atol(group_view_mesg.proc_id);
group_view_mesg.mesg_len = 12 + 22;
gview_mesg_send(id, &group_view_mesg);
```

```
}
else if  (strcmp(group_view_mesg.msg_header, "rankmembr") == 0)  {

    /*
     * read in the members from the list and compare
     * with the member address given. if a match
     * is found then identify the    distance from
     * head pointer as the rank.
     */
    for (m1 = 0; m1 < atoi(group_view_mesg.num_mem); m1++){
        tempptr = headptr;

        /*
         * extract the member address portion from the list and
         * compare with the member address given. If there is
         * no match continue till the complete list is covered.
         */

        for ( loop_index = 0; loop_index<group_view.group_size;
        loop_index++)  {
            m2 = loop_index;
            if (strcmp( tempptr->memb_addr,
            group_view_mesg.mesg_data[m1].member_address)
            != 0) {

              tempptr1 = tempptr;
              tempptr = tempptr1->next;
            }


            else {

        /* If a match is found the loop index gives its rank.
         */
              group_view_mesg.mesg_data[m1].
              member_rank[1] = ((m2%10) +48);
              group_view_mesg.mesg_data[m1].
              member_rank[0] = (((m2%100)- m2%10)/10 +48);

              break;
            }
        }
    }
    /*
     * Send the message back to the client. fill the
     * message structure with all relavant details.
     */

    group_view_mesg.mesg_type = atol(group_view_mesg.proc_id);
    group_view_mesg.mesg_len =
    12 * atoi(group_view_mesg.num_mem) + 22;
```

89

```
            gview_mesg_send(id, &group_view_mesg);

    }

    /*
     * this segment checks for the list as it prints it out.
     */

    tempptr = headptr;

    for (loop_index3 = 0; loop_index3 < group_view.group_size;
    loop_index3++)  {
        printf("check for string %s member rank is %s\n",
        tempptr->memb_addr,
        group_view_mesg.mesg_data[loop_index3].member_rank);

        tempptr1 = tempptr->next;
        tempptr = tempptr1;
    }




}
```

B. TOKENPOOL SERVER

```
#include   <stdio.h>
#include   "tkpmesg.h"
#include   "msgq.h"


   /* This program maintains and manages the tokenpool.
    * it receives messages from clients and acts on them accordingly.
    */



TKPMesg      token_pool_mesg,mesg1;

typedef    struct list_node *MembPtr;

typedef struct list_node {        /* Token pool  structure */
    char     initiator_addr[9];
    char     memb_addr[9];
    char    token_type[10]; /* token type of member   */
    MembPtr     next;
} TokenPoolNode;

struct view {/* the Status table entries*/
    int number_of_entries;
} token_view;
    MembPtr    tempptr, headptr, tailptr,tempptr1;
```

90

```
main()
{
    int    id, prid;
    long   key;
    /*
     * create message queue if required.
     */
    key =1;
    prid= getpid();

    if ( ( id= msgget(TKPSER,( PERMS | IPC_CREAT))) < 0 ) {
        err_sys("server: cant get message queue for tokenpool server");
        }
    /*
     * do it eternally as an iterative server
     */
    while (key == 1)    {
        tokenpool_server(id);
    }

    exit(0);
}


tokenpool_server(id)
int    id;
{
    int    loop_index, loop_index1, loop_index2, loop_index3, grpsize;
    int    grpview_num, num_bytes_read , token_found, number_search;
    long   proc_id_long;
    char   errmesg[256], *sys_err_str();

    /*
     * Read the message from the message queue
     */

  /* type for client to server messages  */

    token_pool_mesg.mesg_type = 1L;

    if ( ( num_bytes_read= tkp_mesg_recv(id,&token_pool_mesg)) <= 0)
        err_sys("server: message read  error");
    proc_id_long =atol(token_pool_mesg.proc_id);

    token_pool_mesg.mesg_type = proc_id_long;


    /*
     * check for the message header part. there are five possibilites
     * If the header says "Tokenstat" then status of token as old
```

91

```
 *  or new is given. if it is "Add_token" the token is
 * added in the end of group.
 * if it is "uptknpool" then the list is traversed till agreement
 * token for that particular member is found. If found all tokens
 * before and inclusive of the agreement token are purged and
 * the commit token is added at the end of token pool.
 * If the header says "tkpoolreq" then the current token
 * pool is sent as a message. If the header says "initkpool"
 * then a list is created with the message supplied.
 */


if (strcmp(token_pool_mesg.msg_header, "Tokenstat") == 0)  {

   /*
    * This segment checks the list to find out if the token
    * is present or not.
    */

   tempptr = headptr;
   for ( loop_index = 0; loop_index < token_view.number_of_entries;
   loop_index++)  {

       if ((strcmp( tempptr->memb_addr,
       token_pool_mesg.mesg_data[0].member_address) == 0) &
       (strcmp( tempptr->token_type,
       token_pool_mesg.mesg_data[0].token_type) == 0) ) {

       token_found = 1;
       break;
       }

       else {
          tempptr = tempptr->next;
       }
   }

   if (token_found == 0) {
       strcpy(token_pool_mesg.msg_header, "notpresnt");
   } else  {
       strcpy(token_pool_mesg.msg_header, "yespresnt");
    }

   token_pool_mesg.mesg_len = 28 +17;
   tkp_mesg_send(id, &token_pool_mesg);

}
else if (strcmp(token_pool_mesg.msg_header, "add_token") == 0)  {

    /* this segment adds token to the end of the list after
     * getting additional allocation.
```

```
        */

        tempptr = (MembPtr) malloc( sizeof( TokenPoolNode));
        tempptr->next = NULL;
        tailptr->next = tempptr;
        tailptr = tempptr;
        strcpy(tempptr->token_type,
        token_pool_mesg.mesg_data[0].token_type);
        strcpy(tempptr->memb_addr,
        token_pool_mesg.mesg_data[0].member_address);
        strcpy(tempptr->initiator_addr,
        token_pool_mesg.mesg_data[0].initiator_address);
        token_view.number_of_entries++;
}

else if (strcmp(token_pool_mesg.msg_header, "uptknpool") == 0)  {

    /*
     * This segment updates the token pool with the commit token.
     * the list is purged to remove tokens upto and inclusive
     * of agree token of the member address .
     */
    tempptr = headptr;
    number_search = 0;

    for (loop_index = 0; loop_index < token_view.number_of_entries;
    loop_index++)  {

        /* searh the list till agreement token for the commit action
         * is found.
         */

        number_search++;

        if ((strcmp( tempptr->memb_addr,
        token_pool_mesg.mesg_data[0].member_address) == 0) &
        (strcmp(token_pool_mesg.mesg_data[0].token_type,
        "joincomit") == 0) &
        (strcmp( tempptr->token_type,
        "joinagree") == 0) ) {

            break;
        }

        else if ((strcmp( tempptr->memb_addr,
        token_pool_mesg.mesg_data[0].member_address) == 0) &
        (strcmp(token_pool_mesg.mesg_data[0].token_type,
        "failcomit") == 0) &
        (strcmp( tempptr->token_type,
        "failagree") == 0) ) {
```

```
        break;
    }

    else {
        tempptr = tempptr->next;
    }
}

/* remove all commit tokens occuring before the commit token  */

tempptr = headptr;
for (loop_index1 = 0; loop_index1 < number_search;
loop_index1++)  {

    if (!((strcmp( tempptr->token_type,
    "joincomit") == 0) |
    (strcmp(tempptr->token_type,
    "failcomit") == 0) |
    ((strcmp( tempptr->memb_addr,
    token_pool_mesg.mesg_data[0].member_address) == 0) &
    (((strcmp(token_pool_mesg.mesg_data[0].token_type,
    "failcomit") == 0) &
    (strcmp( tempptr->token_type,
    "failagree") == 0) ) |
    ((strcmp(token_pool_mesg.mesg_data[0].token_type,
    "joincomit") == 0) &
    (strcmp( tempptr->token_type,
    "joinagree") == 0) ) ) ))) {

        tempptr1 = tempptr;
        tempptr = tempptr1->next;
    }

/*
 * If the match is for the  first member removal
 * update the pointer to the head of the list.
 */

    else if (loop_index1 == 0) {
        tempptr = headptr;
        headptr = tempptr->next;
        token_view.number_of_entries--;
        free(tempptr);
    }

    else {

/* If a match is found in the middle of the list change
 * the link to reform the list.
 */
```

```
            tempptr1->next = tempptr->next;
            token_view.number_of_entries--;
            free(tempptr);
            if (tempptr1->next == NULL)
                tailptr = tempptr1;
        }
    }

    /*
     * add commit token to the end of the list.
     */

    tempptr = (MembPtr) malloc( sizeof( TokenPoolNode));
    tempptr->next = NULL;
    tailptr->next = tempptr;
    tailptr = tempptr;
    strcpy(tempptr->token_type,
    token_pool_mesg.mesg_data[0].token_type);
    strcpy(tempptr->memb_addr,
    token_pool_mesg.mesg_data[0].member_address);
    strcpy(tempptr->initiator_addr,
    token_pool_mesg.mesg_data[0].initiator_address);
    token_view.number_of_entries++;

}


else if (strcmp(token_pool_mesg.msg_header, "initkpool") == 0)  {


    /*
     * this part of the program initialises the
     * list structure and generates the initial token pool.
     */

    token_view.number_of_entries  = atoi(token_pool_mesg.num_mem);

    /*
     * create the list structure. make the
     * pointer to the first element the header
     */

    for (loop_index1 = 0; loop_index1 < token_view.number_of_entries;
    loop_index1++)  {

        tempptr = (MembPtr) malloc( sizeof( TokenPoolNode));
        if (loop_index1 ==0) {
            headptr = tempptr; /* head of the list */
            tailptr = headptr; /* initial tail  */
        }
```

```
        /*
         * copy the number of members iteratively.
         */

        strcpy(tempptr->memb_addr,
        token_pool_mesg.mesg_data[loop_index1].member_address);
        strcpy(tempptr->initiator_addr,
        token_pool_mesg.mesg_data[loop_index1].initiator_address);
        strcpy(tempptr->token_type,
        token_pool_mesg.mesg_data[loop_index1].token_type);
        tempptr->next = NULL;
        tailptr->next = tempptr;
        tailptr = tempptr;
    }
}
else if (strcmp(token_pool_mesg.msg_header ,"Tkpoolreq")== 0 ){

    grpsize = token_view.number_of_entries;


    /*
     * integer to ascii conversion routine for number
     * of members .the string is a null terminated
     */

    itoa(token_pool_mesg.num_mem, grpsize, 3);
    tempptr = headptr;
    for (loop_index2 = 0; loop_index2 < grpsize; loop_index2++) {
        strcpy(
        token_pool_mesg.mesg_data[loop_index2].member_address ,
        tempptr->memb_addr);
        strcpy(
        token_pool_mesg.mesg_data[loop_index2].initiator_address ,
        tempptr->initiator_addr);
        strcpy(token_pool_mesg.mesg_data[loop_index2].token_type ,
        tempptr->token_type);
        token_pool_mesg.mesg_data[loop_index2].member_address[8] =
        NULL;
        tempptr1 = tempptr;
        tempptr = tempptr1->next;
    }

    token_pool_mesg.mesg_type = atol(token_pool_mesg.proc_id);
    token_pool_mesg.mesg_len = grpsize*28 + 17;
    tkp_mesg_send(id, &token_pool_mesg);
}


/*
 * This segment of program checks the list after each change
```

```
    */

    tempptr = headptr;

    for (loop_index3 = 0;loop_index3 <
    token_view.number_of_entries;loop_index3++)  {

        printf("check for string %s\n", tempptr->memb_addr);
        printf("check for string %s\n", tempptr->token_type);
        tempptr1 = tempptr->next;
        tempptr = tempptr1;
    }


}
```

C. STATUS TABLE SERVER

```
#include   <stdio.h>
#include   "stmesg.h"
#include   "msgq.h"

    /* This program maintains and manages the status table.
     * it receives messages from clients and acts on them accordingly.
     */

STMesg      status_table_mesg,mesg1;

typedef   struct list_node *MembPtr;

typedef struct list_node {         /* Status table structure  */
    char    memb_addr[9];
    char    member_status[10]; /* status of member   */
    MembPtr    next;
} StatusTableNode;

struct view {/* the Status table entries*/
    int number_of_entries;
} status_view;
    MembPtr    tempptr, headptr, tailptr,tempptr1;


main()
{
    int    id, prid;
    long    key;
    /*
     * create message queue  for status table server if required.
     */
```

```
    key =1;
    prid= getpid();

    if ( ( id= msgget(STSER,( PERMS | IPC_CREAT))) < 0 ) {
     err_sys("server: cant get message queue for status table server");
        }
    /*
     * do it eternally as an iterative server
     */
    while (key == 1)    {
        status_table_server(id);
    }

    exit(0);
}


status_table_server(id)
int    id;
{
    int    loop_index, loop_index1, loop_index2, loop_index3, grpsize;
    int    grpview_num, num_bytes_read , address_found, m1;
    long   proc_id_long;
    char   errmesg[256], *sys_err_str();

    /*
     * Read the message from the message queue
     */

    /* type for client to server messages  */
    status_table_mesg.mesg_type = 1L;

    if ( ( num_bytes_read= st_mesg_recv(id,&status_table_mesg)) <= 0)
        err_sys("server: message read  error");
    proc_id_long =atol(status_table_mesg.proc_id);

    status_table_mesg.mesg_type = proc_id_long;


    /*
     * check for the message header part. there are seven possibilites
     * if the header says "Statmembr" then the member's status
     * is sent if available. If the header says "updtstats" then
     * the member is added in the end of group with the updated status.
     * If the header says "statblreq" then the current status
     * table is sent as a message. If the header says "inittable"
     * then a list is created with the message supplied.
     * If the header says "removmemb" then the entry for the member
     * is removed from status table. If the header is "checkpend"
     * the status table sends the member address whose status is either
     * failpending or joinpending. For header "getmemadr" the server
     * returns all member address having the same status given in the
```

98

```
 * message.
 */


if  (strcmp(status_table_mesg.msg_header, "checkpend") == 0)  {
   tempptr = headptr;
   status_table_mesg.num_mem[0] = 48;
   status_table_mesg.num_mem[1] = 48;

  .for (loop_index = 0;
   loop_index < status_view.number_of_entries; loop_index++)  {

   /*
    * check if any member in the status table has a pending
    * status. If present send the address. num_mem field specifies
    * the presence or absence of the member having pending status.
    */

       if ((strcmp( tempptr->member_status,"joinpendg") == 0) |
        (strcmp( tempptr->member_status,"failpendg") == 0)) {

           strcpy(status_table_mesg.mesg_data[0].member_status,
           tempptr->member_status);
           strcpy(status_table_mesg.mesg_data[0].member_address,
           tempptr->memb_addr);

           status_table_mesg.num_mem[0] = 49; /* ascii 1 */
           break;
       }
       else {
           tempptr = tempptr->next;
       }
   status_table_mesg.mesg_len = 19 + 17;
   status_table_mesg.mesg_type = atol(status_table_mesg.proc_id);
   st_mesg_send(id, &status_table_mesg);
   }
}

if  (strcmp(status_table_mesg.msg_header, "statmembr") == 0)  {

   /*
    * This segment of program gets the current status of
    * the member.If the member address is not present
    * then it signals in the status field as not present.
    */


   /*
    * read in the member address from message  and compare
    * with the member address in the list . If a match
    * is found then send a message giving the current status.
```

```c
    */

    tempptr = headptr;
    address_found = 0;

    /*
     * search the table for the entry corresponding to the member
     * address given. If the address is found send the status
     * as a message.
     */

    for ( loop_index = 0;
    loop_index < status_view.number_of_entries; loop_index++) {

        if (strcmp( tempptr->memb_addr,
        status_table_mesg.mesg_data[0].member_address) == 0) {

        strcpy(status_table_mesg.mesg_data[0].member_status,
        tempptr->member_status);
        address_found = 1;
        break;
        }

        else {
            tempptr = tempptr->next;
        }
    }

    /*
     * if address is not found send the message as "not Found".
     */

    if (address_found == 0)
        strcpy(status_table_mesg.mesg_data[0].member_status,
        "notpresnt");


    /*
     * send the message to the client. fill the message structure
     * with appropriate data.
     */

    status_table_mesg.mesg_len = 19 + 17;
    status_table_mesg.mesg_type = atol(status_table_mesg.proc_id);

    st_mesg_send(id, &status_table_mesg);
}

else if (strcmp(status_table_mesg.msg_header, "Updtstats") == 0) {

    /*
```

```
 * This segment updates the status of the member
 * if the member is already present it updates the status
 * otherwise it adds the member and status in the tail of
 * list.
 */
tempptr = headptr;
address_found = 0;
for ( loop_index = 0;
loop_index < status_view.number_of_entries; loop_index++)  {

    if (strcmp( tempptr->memb_addr,
    status_table_mesg.mesg_data[0].member_address) == 0) {

    strcpy(tempptr->member_status,
    status_table_mesg.mesg_data[0].member_status);
    address_found = 1;
    break;
    }

    else {
        tempptr = tempptr->next;
    }
}

/*
 * if member is not already present add the member with
 * status specified.
 */

if (address_found == 0) {
    tempptr = (MembPtr) malloc( sizeof( StatusTableNode));
    tempptr->next = NULL;
    tailptr->next = tempptr;
    tailptr = tempptr;
    strcpy(tempptr->memb_addr,
    status_table_mesg.mesg_data[0].member_address);
    strcpy(tempptr->member_status,
    status_table_mesg.mesg_data[0].member_status);
    status_view.number_of_entries++;
}

}
else if (strcmp(status_table_mesg.msg_header, "inittable") == 0)  {

    /*
     * this part of the program initialises the
     * list structure and generates the initial status table.
     */

    status_view.number_of_entries  = atoi(status_table_mesg.num_mem);
```

101

```
/*
 * create the list structure. make the
 * pointer to the first element the header
 */

for (loop_index1 = 0; loop_index1 < status_view.number_of_entries;
loop_index1++)  {


    tempptr = (MembPtr) malloc( sizeof( StatusTableNode));
    if (loop_index1 ==0) {
       headptr = tempptr; /* head of the list */
        tailptr = headptr; /* initial tail  */
    }

    /*
     * copy the number of members iteratively.
     */

    strcpy(tempptr->memb_addr,
    status_table_mesg.mesg_data[loop_index1].member_address);
    strcpy(tempptr->member_status,
    status_table_mesg.mesg_data[loop_index1].member_status);
    tempptr->next = NULL;
    tailptr->next = tempptr;
    tailptr = tempptr;
  }
}
else if(strcmp(status_table_mesg.msg_header ,"getmemadr")== 0 ){

  grpsize = status_view.number_of_entries;


  /* create the message from the list. Fill all the
   * other data required for the message to be sent.
   */

  tempptr = headptr;
  m1 = 0;

   /*
    * check the list to see if there is a match with the
    * status given in the message to the status of members
    * in the list. If a match is found add them to the meesage
    * and increment the number of items in the message.
    */

   for (loop_index2 = 0; loop_index2 < grpsize; loop_index2++) {
     if (strcmp( tempptr->member_status,"failagree") == 0){
        strcpy(status_table_mesg.mesg_data[m1].member_address ,
```

```c
            tempptr->memb_addr);
            strcpy(status_table_mesg.mesg_data[m1].member_status ,
            tempptr->member_status);
            m1++;
            tempptr = tempptr->next;
        }
        else  {
            tempptr = tempptr->next;
        }
    }

    status_table_mesg.num_mem[1] = m1%10 +48;
    status_table_mesg.num_mem[0] = (m1%100-m1%10)/10 +48;
    status_table_mesg.num_mem[2] = NULL;
    status_table_mesg.mesg_type = atol(status_table_mesg.proc_id);
    status_table_mesg.mesg_len = m1*19 + 17;
    st_mesg_send(id, &status_table_mesg);
}


else if(strcmp(status_table_mesg.msg_header ,"statblreq")== 0 ){

    grpsize = status_view.number_of_entries;


    /*
     * integer to ascii conversion routine for number
     * of members .the string is a null terminated
     */

    status_table_mesg.num_mem[1] = grpsize%10 +48;
    status_table_mesg.num_mem[0] = (grpsize%100-grpsize%10)/10 +48;
    status_table_mesg.num_mem[2] = NULL;

    /* create the message from the list. Fill all the
     * other data required for the message to be sent.
     */

    tempptr = headptr;
    for (loop_index2 = 0; loop_index2 < grpsize;
    loop_index2++) {
        strcpy(
        status_table_mesg.mesg_data[loop_index2].member_address ,
        tempptr->memb_addr);
        strcpy(
        status_table_mesg.mesg_data[loop_index2].member_status ,
        tempptr->member_status);
        status_table_mesg.mesg_data[loop_index2].member_address[8] =
        NULL;
        printf(" running in loop");
        tempptr1 = tempptr;
```

```
            tempptr = tempptr1->next;
    }

    status_table_mesg.mesg_type = atol(status_table_mesg.proc_id);
    status_table_mesg.mesg_len = grpsize*19 + 17;
    printf("message length =%d\n",status_table_mesg.mesg_len);
    st_mesg_send(id, &status_table_mesg);
}


else if (strcmp(status_table_mesg.msg_header, "removmemb") == 0) {

    /*
     * read in the members from the list and compare
     * with the member to be removed. if a match
     * is found then remove the member .
     * change the link address so that the link is not broken
     */

    tempptr = headptr;

    /*
     * extract the member address portion from the list and
     * compare with the member to be removed. If there is no match
     * go to the next member till the complete list is covered.
     */

    for (loop_index = 0; loop_index < status_view.number_of_entries;
    loop_index++)  {

        if (strcmp( tempptr->memb_addr,
        status_table_mesg.mesg_data[0].member_address) != 0) {

            tempptr1 = tempptr;
            tempptr = tempptr1->next;
        }

    /*
     * If the match is for the host( first member) remove host
     * and update the pointer to the list
     */

        else if (loop_index == 0) {
            tempptr = headptr;
            headptr = tempptr->next;
            free(tempptr);
            break;
        }

        else {
```

```
        /* If a match is found in the middle of the list change
         * the link to reform the list.
         */

            tempptr1->next = tempptr->next;
            free(tempptr);
            if (tempptr1->next == NULL)
                tailptr = tempptr1;
            break;
        }
    }

    }

    tempptr = headptr;

    for (loop_index3 = 0; loop_index3 < status_view.number_of_entries;
    loop_index3++)  {

        printf("check for string %s\n", tempptr->memb_addr);
        printf("check for string %s\n", tempptr->member_status);
        tempptr1 = tempptr->next;
        tempptr = tempptr1;
    }



}


D.  COMMIT PROCESS SERVER
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include     "commesg.h"
#include     "tkpmesg.h"
#include     "gvmesg.h"
#include     "stmesg.h"
#include     "injoinmesg.h"
#include     "msgq.h"

TKPMesg      tkpmesg;
GVMesg       gvmesg;
STMesg       stmesg;
INJOINMesg    injoin_mesg;
COMMesg commit_mesg;
main()
{
    int     comid, key;

    /*
```

```
        * create and open  the single message queue for commit processing.
        */

     if ( ( comid= msgget(COMMITQ,(PERMS | IPC_CREAT ))) < 0 )
         err_sys("Commit_process: can't msgget message queue ");

     key = 1;

     if ( key == 1)   {
     Commit_process(comid);

     }
     exit(0);
}

Commit_process(comid)
int comid;
{
    int n,n1,i, rank_member, m1,rankpend;
    char c1, temp[MAXMESGDATA],host_address[9],pending_member[9];
    char my_address[9], pending_status[10];
     int    tkpid,stid,gvid,commit_pend_process, new_token;

    /*
     * open the message queues for groupview server, tokenpool server
     * and status table server.
     */

    if ( ( tkpid= msgget(TKPSER, 0 )) < 0 )
        err_sys("Commit_process: can't msgget tokenpool server queue ");
    if ( ( gvid= msgget(GVSER, 0 )) < 0 )
        err_sys("Commit_process: can't msgget groupview server queue ");
    if ( ( tkpid= msgget(STSER, 0 )) < 0 )
        err_sys("Commit_process: can't msgget stat table server queue");
/*   if ( ( imjoinid= msgget(INJOINSER, 0 )) < 0 )
 *       err_sys("Commit_process: can't msgget initiate join queue");
/*
 * wait for commit initiate message.
 */
    new_token = TRUE;
    commit_pend_process = FALSE;
    commit_mesg.mesg_type = 1;
    n= commit_mesg_recv(comid, &commit_mesg);
    if (n < 0 )
        err_sys( "data read error");
    while (new_token | commit_pend_process) {

     commit_pend_process = FALSE;
     new_token = FALSE;
     /*
      * send a query to token pool server to see if it a old token
```

```
     * fill in appropriate details for tokenpool server can
     * receive the message
     */

   n1 = getpid();

   /* fill in the process id field in the message     */

   itoa(tkpmesg.proc_id , n1, 5);

/* fill in the message header and numbers field in the message */

   strcpy(tkpmesg.msg_header, "tokenstat");
   strcpy(tkpmesg.num_mem, "01");

   /* fill in the token details  in the message     */

   strcpy(tkpmesg.mesg_data[0].token_type,
   commit_mesg.mesg_data.token_type);
   strcpy(tkpmesg.mesg_data[0].initiator_address,
   commit_mesg.mesg_data.initiator_address);
   strcpy(tkpmesg.mesg_data[0].member_address,
   commit_mesg.mesg_data.member_address);

   tkpmesg.mesg_len=(28 + 17);
   tkpmesg.mesg_type= 1L;

   tkp_mesg_send(tkpid, &tkpmesg);

   /* receive reply  from tokenpool server */

   tkpmesg.mesg_type = n1%10000;
   n= tkp_mesg_recv(tkpid, &tkpmesg);
   if (n < 0 )
       err_sys( "data read error");

   if (strcmp(tkpmesg.msg_header, "notpresnt"))  {

       /*
        * send update tokenpool message to tokenpool server.
        */

       strcpy(tkpmesg.msg_header, "uptknpool");
       tkp_mesg_send(tkpid, &tkpmesg);

       /*
        * send update status table message to statustable server.
        */

       strcpy(stmesg.msg_header, "removmemb");
```

```
/* fill in the process id field in the message    */

itoa(stmesg.proc_id,  n1, 5);

/* fill in the other fields in the message    */

strcpy(stmesg.num_mem, "01");
strcpy(stmesg.mesg_data[0].member_address,
commit_mesg.mesg_data.member_address);
strcpy(stmesg.mesg_data[0].member_status, "removmemb");
stmesg.mesg_len=(19 + 17);
stmesg.mesg_type= 1L;

st_mesg_send(stid, &stmesg);

/*
 * send update groupview message to groupview server.
 */

if( strcmp(commit_mesg.mesg_data.token_type,"joincomit") == 0)
    strcpy(gvmesg.msg_header, "Upadgview");
else
    strcpy(gvmesg.msg_header, "Uprmgview");

/* fill in the process id field in the message    */

itoa(gvmesg.proc_id, n1, 5);

/* fill in the other fields in the message    */

strcpy(gvmesg.num_mem, "01");
strcpy(stmesg.mesg_data[0].member_address,
commit_mesg.mesg_data.member_address);
strcpy(gvmesg.view_num, "0000");
gvmesg.mesg_len=(12 + 22 );
gvmesg.mesg_type= 1L;

gview_mesg_send(gvid, &gvmesg);

/*
 * update the host address of the group.
 */

strcpy(gvmesg.msg_header, "hostaddrs");
gview_mesg_send(gvid, &gvmesg);

gvmesg.mesg_type = n1%10000;
n = gview_mesg_recv(gvid, &gvmesg);

if (n < 0)
      err_sys("data read error");
```

108

```c
        strcpy(host_address, gvmesg.mesg_data[0].member_address);

/* the way to send tokens and updating the anticlockwise member
 * is pending still
 */

        /* check to see if the running process is the host  */

/*        if(( strcmp(commit_mesg.mesg_data.token_type,"joincomit")) &
 *        ( strcmp(host_address, my_address)))  {
 *
 *            /*
 *             * fill initiate join message with appropriate
 *             * values.
 *             */
/*            strcpy(injoin_mesg.msg_header,"sendinitp");
 *            injoin_mesg.mesg_type = 1L;
 *            strcpy(injoin_mesg.member_address,
 *            commit_mesg.mesg_data.member_address);
 *            injoin_mesg.mesg_len = 20;
 *            injoin_mesg_send(injoinid, &injoin_mesg);
 *        }
*/       /*
         * send check commit pending message to statustable server.
         */

        strcpy(stmesg.msg_header, "checkpend");

        /* fill in the process id field in the message    */

        itoa(stmesg.proc_id , n1, 5);

        /* fill in the other fields in the message    */

        strcpy(stmesg.num_mem, "01");
        strcpy(stmesg.mesg_data[0].member_address,
        commit_mesg.mesg_data.member_address);
        strcpy(stmesg.mesg_data[0].member_status, "failpendg");
        stmesg.mesg_len=(19 + 17);
        stmesg.mesg_type= 1L;

        st_mesg_send(stid, &stmesg);

        stmesg.mesg_type = n1%10000;
        n= st_mesg_recv(stid, &stmesg);
        if (n < 0 )
            err_sys( "data read error");


        if (atoi(stmesg.num_mem) != 0) {
            commit_pend_process = TRUE;
```

```
            strcpy(pending_member,stmesg.mesg_data[0].member_address);
            strcpy(pending_status,stmesg.mesg_data[0].member_status);
            strcpy(gvmesg.mesg_data[0].member_address,
            pending_member);
            strcpy(gvmesg.msg_header,"rankmembr");
            gvmesg.mesg_type = 1L;
            gview_mesg_send(gvid,&gvmesg);
            gvmesg.mesg_type = n1%10000;
            n= gview_mesg_recv(gvid, &gvmesg);
            if (n < 0 )
                err_sys( "data read error");
            rankpend = atoi(gvmesg.num_mem);


    /* send a message to get the all agreement tokens   */

      strcpy(stmesg.msg_header, "getmemadr");


    /* fill in the other fields in the message    */

            strcpy(stmesg.num_mem, "01");
            strcpy(stmesg.mesg_data[0].member_address,
            commit_mesg.mesg_data.member_address);
            strcpy(stmesg.mesg_data[0].member_status,
            "failagree");
            stmesg.mesg_len=(19 + 17);
            stmesg.mesg_type= 1L;

            st_mesg_send(stid, &stmesg);

    /* get a list of members who are agreed on failing   */

            stmesg.mesg_type = n1%10000;
            n= st_mesg_recv(stid, &stmesg);
            if (n < 0 )
                err_sys( "data read error");



            /*
             * this segment checks if there are other agree
             * for failures member present.
             */

            if ( !(strcmp(stmesg.num_mem,"00") ==0 )) {
                /*
                 * this segment gets the rank of all
                 * members with a failure agree.
                 */
```

110

```
                    for( m1 = 1; m1 <= atoi(stmesg.num_mem); m1++ ){
                      strcpy(gvmesg.mesg_data[m1].
                      member_address,stmesg.mesg_data[m1].
                      member_address);
                    }
                    strcpy(gvmesg.msg_header,"rankmembr");
                    gvmesg.mesg_type = 1L;
                    gview_mesg_send(gvid,&gvmesg);

                    gvmesg.mesg_type = n1%10000;
                    n= gview_mesg_recv(gvid, &gvmesg);
                    if (n < 0 )
                     err_sys( "data read error");

                    for( m1 = 1; m1 <= atoi(stmesg.num_mem); m1++ ){
                      rank_member= atoi(gvmesg.mesg_data[m1].member_rank);

                      if( rank_member < rankpend)  {
                          commit_pend_process = FALSE;
                          break;
                      }
                    }
                }
            }

        if (commit_pend_process) {
            strcpy(commit_mesg.mesg_data.member_address,
            pending_member);
            strcpy(commit_mesg.mesg_data.initiator_address,
            my_address);
            if ( strcmp(pendıng_status, "joinpendg") == 0)
                strcpy(commit_mesg.mesg_data.token_type,"joincomit");
            else
                strcpy(commit_mesg.mesg_data.token_type,"failcomit");

        }


    }
  }
}


/*
 *     send a message by using SysTEm V message queues.
 *     The mesg_len, mesg_type and mesg_data must be filled in by the
 *     caller
 */

commit_mesg_send(id,mesgptr)
int     id;
```

111

```
COMMesg    *mesgptr;
{


    /*
     * Send the message - the type followed by the optional data.
     */

    if (msgsnd(id,(char *) &(mesgptr->mesg_type),
    mesgptr->mesg_len, 0) != 0)
        err_sys("msgsend error");

}


/*
 *    receive a message by reading on a file descriptor.
 *    fill in the mesg_len, mesg_type and mesg_data also
 */


int
commit_mesg_recv(id, mesgptr)
int    id;
COMMesg    *mesgptr;

{

    int    n;

    n = msgrcv(id,  &(mesgptr->mesg_type),MAXMESGDATA,
    mesgptr->mesg_type, 0);

    if ( (  mesgptr->mesg_len = n ) < 0)
            err_sys("msgrcv error");
    return(n);
}
```

REFERENCES

[BSS91]   K. Birman, A. Schiper. and P. Stephenson. Lightweight causal and atomic group multicast. In *ACM Transactions on Computer Systems*, pages 272-314, 1991.

[B+90]   K. Birman et al. *ISIS - A distributed Programming Environment (Programmers Manual)*. Department of Computer Science, Cornell University, August 1990. Rev 2.1.

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

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

[CM89]   Douglas Comer. Internetworking with TCP/IP. Prentice Hall software series. 1989.

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

[Cri88]   F. Cristain. 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.a

[CT90]   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.

[DSS1]   S. Mullander. Distributed Systems. ACM press. 1990.

[EzLe90] P. D. Ezhilselvan and Rogerio de Lomos. A robust group membership algorithm for distributed real-time systems. In *Proceedings Real-Time Systems Symposium*, pages 173-179, 1990

[FLP85]   M.J. Fischer, N.A. Lynch, and M.S. Peterson. Impossibility of distributed consensus with one faulty process. *Journal of the Assosiation of Computing Machinery*, pages 374-382, April 1985.

[LSA91]   L.E. Moser, P.M. Melliar Smith, V. Agrawala. Membership algorithm for asynchronous distributed systems. In *Proceedings of the Eleventh International Conference on Dependable Computing for Critical Applications, Santa Barbara, CA* , Pages 167-174, 1989.

[RB91]   A. Ricardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. Technical Report TR91-1188, Cornell University, February 1991.

[Roch]   Steve Rochkind.Advanced UNIX programming. Prentice Hall software se-
            ries. 1987.

[ShDr]   S. Shukla and R. Devalla. Group Membership in Asynchronous Distributed
            Systems using Logically Ordered Groups. Technical Report NPSEC-92-009
            . Naval Postgraduate School. 1992

[SR90]   W.R. Stevens. UNIX Network Programming. Prentice Hall software series.
            1990.

[SUN]   SUN. Network programming manual for SUNOS 4.1. 1990.

[STP]   Software Through Pictures User's Manual. Interactive Development Envi-
            ronment. San Francisco 1990.

INITIAL DISTRIBUTION LIST

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

2. Director, Directorate of Training and Sponsored Research   4
    Defense Research and Development Organisation.
    Ministry of Defense
    227 'B' Block . Sena Bhavan
    New Delhi, INDIA 110011

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

4. Director   2
    Defense Electronics Research Laboratory.
    Chandrayangutta Lines
    Chandrayangutt.
    Hyderabad, INDIA 500005

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

6. Chairman, Electronic Warfare Academic Group.   1
    Code EW
    Naval Postgraduate School
    Monterey, CA 93943

7.   Dr. Shridhar B Shukla   1
      Code EC/Sh
      Naval Postgraduate School
      Monterey, CA 93943

8.   Dr. Douglas J Fouts   1
      Code EC/Fs
      Naval Postgraduate School
      Monterey, CA 93943