

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A267 309



DTIC
SELECTE
JUL 28 1993
S B D

THESIS

DESIGN OF A DECENTRALIZED ASYNCHRONOUS
MEMBERSHIP PROTOCOL AND AN IMPLEMENTATION
OF ITS COMMUNICATIONS LAYER

by

Fernando Jorge Pires

March 1993

Thesis Advisor:

Shridhar B. Shukla

Approved for public release; distribution is unlimited.

93-16901



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 Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) Code 32	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
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.	TASK NO.
		PROJECT NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) DESIGN OF A DECENTRALIZED ASYNCHRONOUS MEMBERSHIP PROTOCOL AND IMPLEMENTION OF ITS COMMUNICATIONS LAYER			
12. PERSONAL AUTHOR(S) Fernando Jorge Pires			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 06/92 TO 03/93	14. DATE OF REPORT (Year, Month, Day) March 1993	15. PAGE COUNT 155
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	Agreement, Asynchronous, Commit, Decentralized, Failure, Fault-tolerance, Group membership, Join, Logical ring, Reliable multicast, Token	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) For development of group-oriented distributed applications, a group membership protocol provides the mechanisms to dynamically adapt to changes in the membership, ensuring consistent views among all members of the group. This is achieved, by executing a distributed script, that implements a protocol, at each member to maintain a sequence of identical views, in spite of continuous changes, either voluntary or due to failure, to the membership. In asynchronous distributed environments, the protocol has to operate over a network that does not bound delivery times. This thesis presents a decentralized membership protocol, designed to operate on asynchronous environments, that organizes the members in a logical ring. The protocol assumes reliable FIFO channels, that fully interconnect all members to be available. These assumptions are later relaxed to adapt the protocol to real-world environments. Reconfigurations of the group are carried out using a two-phase algorithm. An agreement phase makes the change known to all operational members, and a commit phase integrates the change at all members, in the correct order. The protocol supports failures of one or more members, either successive or simultaneous, voluntary departures, and joining of new members. In the case of simultaneous events, the protocol ensures that they are incorporated one at a time, and fol-			
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) 656-2764	22c. OFFICE SYMBOL EC/Sh

lowing the same sequence, at all members. All actions are token-based and the protocol ensures that no tokens are lost or duplicated regardless of changes in the membership during any phase of the protocol. The main feature of this protocol is that, by ordering the group in a logical ring, and by decentralizing the responsibility of the monitoring and re-configuration processes, the need for a dedicated manager is eliminated. Execution of the protocol is symmetric relative to the type of change, and to the responsibility distribution among members. The complete specification of the protocol is presented, along with a correctness proof, and performance analysis. A full implementation design is presented and the actual implementation (coding) issues for a Unix-based environment are discussed. Since there are no other known full implementations of a decentralized protocol, comparisons are made with a centralized protocol, to determine message cost, and scalability characteristics.

Accession For		☒	☐	☐
NTIS GRA&I				
DTIC TAB				
Unannounced				
Justification				
By				
Distribution/				
Availability Codes				
Avail and/or				
Special				
Dist				
A-1				

DTIC QUALITY INSPECTED 5

Approved for public release; distribution is unlimited

**DESIGN OF A DECENTRALIZED ASYNCHRONOUS
GROUP MEMBERSHIP PROTOCOL
AND AN IMPLEMENTATION OF ITS COMMUNICATIONS LAYER**

by

Fernando Jorge Pires
Lieutenant, Portuguese Navy
B.S.E.E., Escola Naval, Lisbon, Portugal , 1986

Submitted in partial fulfillment of the
requirements for the degrees of

ELECTRICAL ENGINEER
and
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

March 1993

Author:

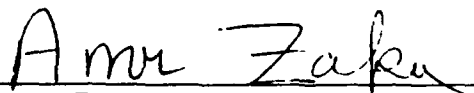


Fernando Jorge Pires

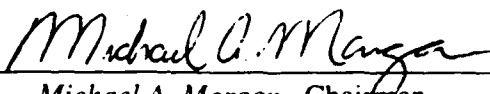
Approved By:



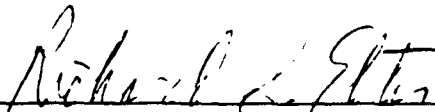
Shridhar B. Shukla, Thesis Advisor



Amr Zaky, Second Reader



Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering



Richard S. Elster,
Dean of Instruction

ABSTRACT

For development of group-oriented distributed applications, a group membership protocol provides the mechanisms to dynamically adapt to changes in the membership, ensuring consistent views among all members of the group. This is achieved, by executing a distributed script that implements a protocol at each member to maintain a sequence of identical views, in spite of continuous changes, either voluntary or due to failure, to the membership. In asynchronous distributed environments, the protocol has to operate over a network that does not bound delivery times. This thesis presents a decentralized membership protocol, designed to operate on asynchronous environments, that organizes the members in a logical ring. The protocol assumes reliable FIFO channels, that fully interconnect all members to be available. These assumptions are later relaxed to adapt the protocol to real-world environments. Reconfigurations of the group are carried out using a two-phase algorithm. An agreement phase makes the change known to all operational members, and a commit phase integrates the change at all members, in the correct order. The protocol supports failures of one or more members, either successive or simultaneous, voluntary departures, and joining of new members. In the case of simultaneous events, the protocol ensures that they are incorporated one at a time, and following the same sequence, at all members. All actions are token-based and the protocol ensures that no tokens are lost or duplicated regardless of changes in the membership during any phase of the protocol. The main feature of this protocol is that, by ordering the group in a logical ring, and by decentralizing the responsibility of the monitoring and reconfiguration processes, the need for a dedicated manager is eliminated. Execution of the protocol is symmetric relative to the type of change, and to the responsibility distribution among members. The complete specification of the protocol is presented, along with a correctness proof and performance analysis. A full implementation design is presented and the actual implementation (coding) issues for a Unix-based environment are discussed. Since there are no other known full implementations of a decentralized protocol, comparisons are made with a centralized protocol, to determine message cost, and scalability characteristics.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND	1
B.	SOLUTIONS	2
C.	SCOPE AND ORGANIZATION OF THE THESIS	4
II.	A DECENTRALIZED GROUP MEMBERSHIP PROTOCOL	6
A.	GROUP MEMBERSHIP PROTOCOL OVERVIEW	6
1.	Assumptions	6
2.	Overview	7
a.	Processing of Individual Changes	9
3.	Definitions	10
a.	Group Membership Problem	11
b.	Logical Ring	11
c.	Ring Relation (RR)	11
d.	Ring Property	12
e.	Logical Marker	12
f.	Ring Host	12
g.	Rank	12
h.	Monitoring Member	12
i.	Tokens	12

j. Local Information	13
k. Neighbor and Host Computation	14
B. THE GROUP MEMBERSHIP PROTOCOL	16
1. Status Change Detection	16
2. The Agreement Phase	19
3. The Commit Phase	21
4. Ensuring An Identical Sequence Of Commits	23
C. SAFETY AND LIVENESS	24
III. PROTOCOL DESIGN	28
A. PROTOCOL SOFTWARE DESIGN	28
B. PROCESS SPECIFICATION	29
1. Fifo Channel Layer	30
a. Front Sub-process	30
b. Back Sub-process	33
2. Monitor Process	35
a. Status Monitor	37
b. Status Reporter	37
c. Timer	38
3. Join Processor	38
4. Integrate Member	40
5. Agreement Processor	41

6. Commit Processor	43
7. Group View Manager	45
8. Status Table Manager	46
9. Token Pool Manager	47
C. DATA STRUCTURE DEFINITIONS AND MESSAGE FORMATS	48
1. Data structure definition	48
a. Group View	48
b. Status Table	50
c. Token Pool	50
2. Message formats	51
a. Token message	54
b. Token Pool message	55
c. Initial Token Pool	56
d. Token Ack message	57
e. Delete Token message	57
f. Initiate Token message	58
g. Status Table and Initial Status Table messages	58
h. Group View and Initial Group View messages	59
i. Initial Parameters message	60
j. Join Request, Status Query and Status Report messages	61
k. Update Status and Update View messages	62

l. Send Initial Parameters message	63
m. View Request, Status Table Request and Token Pool Request messages ...	63
n. Start Timer and Timeout messages	63
IV. IMPLEMENTATION ON UNIX-BASED MACHINES	65
A. COMMUNICATIONS LAYER	65
1. Network Access Protocol	65
a. Inter-member communications	67
b. Intra-member communications	68
2. Socket Access Abstraction	68
a. Unix-domain socket access	69
b. Internet-domain socket access	71
c. Multi-port socket access	72
B. APPLICATION INTERFACE AND HIERARCHICAL STRUCTURE	73
V. PERFORMANCE AND EXTENSION ANALYSIS	76
A. MP OVER A LAN	76
B. MP OVER A WAN	78
C. STRING OF MEMBERSHIP CHANGES	80
VI. CONCLUSIONS AND RECOMENDATIONS	81
LIST OF REFERENCES	83
APPENDIX	85
INITIAL DISTRIBUTION LIST	140

LIST OF TABLES

Table 1 :	INTERPRETATION OF $ST_{p_i}(p_i)$	14
Table 2 :	PROTOCOL PROCESSES	29
Table 3 :	MESSAGE LIST	53
Table 4 :	TOKEN TYPES	55
Table 5 :	STATUS TYPES	59
Table 6 :	COMPARISON OF SOCKETS AND TLI (ADAPTED FROM [20])	66
Table 7 :	COMPARISON OF FEATURES FOR UDP AND TCP (FROM [20])	67
Table 8 :	MESSAGE COST COMPARISON	77

LIST OF FIGURES

Figure 1 :	A Logical Ring	8
Figure 2 :	MP Interaction With the External World	16
Figure 3 :	Monitoring and Agreement Initiation Actions	17
Figure 4 :	Reporting of the Member Status	17
Figure 5 :	Processing of a Join Request Message/Token	18
Figure 6 :	Processing of Agreement Tokens	20
Figure 7 :	Generate/Receive and Process a Commit Token	21
Figure 8 :	Actions for Committing a Change	22
Figure 9 :	Membership Protocol Interfaces	28
Figure 10 :	Fifo Channel - Process Dependencies	31
Figure 11 :	Fifo Channel - Front Process	32
Figure 12 :	Fifo Channel - Back Process	35
Figure 13 :	Monitor Process - Internal Structure and Dependencies	36
Figure 14 :	Join Processor - Process Dependencies	39
Figure 15 :	Integrate Member - Process Dependencies	41
Figure 16 :	Agreement Processor - Process Dependencies	42
Figure 17 :	Commit Processor - Process Dependencies	44
Figure 18 :	Group View Manager- Process Dependencies	45
Figure 19 :	Status Table Manager- Process Dependencies	46

Figure 20 :	Token Pool Manager- Process Dependencies	47
Figure 21 :	Group View Data Structure	49
Figure 22 :	Status Table Structure	50
Figure 23 :	Token Pool Structure	51
Figure 24 :	Internal Message Format	52
Figure 25 :	External Message Format	52
Figure 26 :	Element Name Structure	54
Figure 27 :	Token Message Format	55
Figure 28 :	Token Pool Message External Format	56
Figure 29 :	Token Pool Message Internal Format	56
Figure 30 :	Initial Token Pool Message Format	57
Figure 31 :	Token Ack Message Format	57
Figure 32 :	Delete Token Message Format	58
Figure 33 :	Initiate Token Message Format	58
Figure 34 :	Status Table and Initial Status Table Message Format	59
Figure 35 :	Group View and Initial Group View Message Format	60
Figure 36 :	Initial Parameters Message Internal Format	61
Figure 37 :	Join Request, Status Query and Status Report Message Formats	62
Figure 38 :	Update Status and Update View Message Format	62
Figure 39 :	Send Initial Parameters Message Format	63
Figure 40 :	View Request, Status Table Request and Token Pool Request Message Format	63

Figure 41 :	Start Timer and Timeout Message Format	64
Figure 42 :	Sequence of Calls to Establish a Working Unix-domain Socket	69
Figure 43 :	Sequence of Calls to Connect to a Remote Unix Socket	70
Figure 44 :	Sequence of Calls to Establish a Working Internet Socket	71
Figure 45 :	Sequence of Calls to Connect to a Remote Internet Socket	72
Figure 46 :	Sequence of Actions to Create a Multi-port Facility	73
Figure 47 :	Hierarchical Program Execution Structure	74
Figure 48 :	MP Over a Single LAN	77
Figure 49 :	MP Over a WAN	79

ACKNOWLEDGMENT

I would like to thank the Portuguese Navy for giving me the opportunity to attend the program of M.S.E.E. and additionally the program of Electrical Engineer at the Naval Postgraduate School.

Although this thesis bears my name, there are a number of others whose contributions I regard as invaluable for its completion. I am specially grateful to my advisor Prof. Shridhar B. Shukla for his constant guidance, availability and open cooperation, in all stages of the work. I would like to thank David Pezdirtz for his cooperation in the coding process.

Lastly, I would like to thank my wife Julieta, for her support and for coping with the hardship of a long separation. To her, I dedicate this thesis.

I. INTRODUCTION

A. BACKGROUND

Distributed computing systems have become prevalent in a wide variety of application fields such as distributed database contexts, real-time settings, and distributed control applications [5]. In contemporary applications, reliability is a major issue, even in non-distributed systems where correctness, fault tolerance, self-management, real-time responsiveness, protection and security are of central concern. Distributed systems add one more layer of complexity, caused by the dispersion of the computation that gives rise to inter-component communications. In these systems, one has to be concerned not only with the behavior of isolated components, but also with the joint behavior of a plural set of components working in a common application context [4].

Of all characteristics of a robust distributed application, fault tolerance is arguably the hardest to implement, and the most desirable for critical applications. In distributed systems, fault tolerance is achieved by replicating information over several processing elements, and providing mechanisms to maintain global consistency in the presence of disturbances [2]. Thus components of a fault-tolerant distributed system depend on reliable communications over the available physical networks. Incorporating such capabilities in each application is a complex task, and penalizes the design of correct and efficient programs. It is then desirable to have high-level communication primitives that facilitate the writing of critical distributed programs. These primitives will allow the abstraction of the network, such that the application does not have to be concerned with inter-process communication management. They also provide the group, as a whole, with self-recovery capabilities from disturbances [14]. Such primitives make an inherently unreliable communication mechanism reliable.

Group-oriented distributed computation, based on reliable communication primitives, has been shown to be an excellent paradigm for developing such robust distributed applications [4][12]. This paradigm relies on groups of entities that cooperate to carry out the computation. A group may

correspond to a set of processes that must behave consistently to provide a service, or a set of processes in which each must determine its function based on which other processes are operational [11]. Therefore, information about the membership of a group is required at all members, in spite of the changes that may occur when members *fail* or *leave* the group, and when they *recover* or *join* the group [1]. Some form of consensus on group membership is necessary. Without it, a member that respects its specification may nonetheless behave inconsistently with respect to another member that has simply seen different group members [5].

A mechanism that ensures consistent views of the membership of a group is basic to the construction of applications that follow this paradigm [1]. The main features of the process group approach are *failure atomicity* for multicast (all-or-nothing policy in multicast delivery even in the presence of failures), and *membership atomicity* for failures and joins to a group (group membership changes are totally ordered and synchronized within all members) [3][9]. In addition there is the need to ensure that the application consists of *safe* and *live* algorithms [5].

These characteristics are provided by reliable primitives, which lead to the requirement that all members of a group incorporate the changes in membership in the same sequence and in a globally consistent way.

The Group Membership Problem (GMP) is the problem of agreeing on the membership of the group, and consistently sharing that information among all members of a given group.

B. SOLUTIONS

Solving the GMP depends critically on whether it is being considered in a *synchronous* or *asynchronous* distributed system.

In synchronous systems, tight synchronization among the clocks of the interacting processes (thus relying on a *global time*) and/or known upper bounds on message delivery times, are used as foundations for the protocol mechanisms. It is then possible to process changes in a serialized and ordered manner, by completely integrating each change one at a time, and following the same sequence, at all members. Application messages wait until changes to membership are completed, and membership changes wait until all pending messages are sent [1].

In asynchronous environments, there is no relationship among clocks of the interacting processes and message delivery times are unbounded. An important consequence is that crashes are indistinguishable from communication delays or slow members. Rather than detecting and ascertaining failures, these can only be *perceived*. It is necessary that members perceived to have failed be removed from the group. In asynchronous systems, this is the only alternative since it has been proved that it is impossible to reach consensus on a failure [13]. The basic function of a Membership Protocol (MP) in such an environment is to ensure that all operational members commit perceived changes to their local views consistently. The consistent commit entails agreement about the change perceived.

Several MPs have been proposed for asynchronous systems.

In [15], failure/recovery detection and notification are achieved using successive message rounds. The number of messages required scales nonlinearly with the group size, and correct recovery requires *a priori* knowledge of the potential members.

Several MPs are proposed in [16] based on total ordering of messages. Such ordering has a high overhead cost and assumes a fault-tolerant, reliable broadcast communications protocol.

In [17], reliable broadcasts are supported by rotating a membership list (token-list) among operational members. When a member holding the token list fails, a reformation phase is entered which guarantees that a single new token-list is generated and committed to by all members. During this phase, normal message traffic is suspended and handling of changes needs an extension to the protocol.

In [7], a fault-tolerant extension to the Unix operating system is presented. It uses a three-way atomic message transmission to coordinate the recovery from single failures. It implements a software-based recovery process for arbitrary programs, and is specially suited for transaction-based applications, where real-time operation is not critical.

A two-phase site-view management protocol to support higher level fault-tolerant communication primitives is presented in [18]. Its drawback of blocking during continuous failures and recoveries is removed in the formal solution proposed in [5]. Assuming a completely connected network of reliable FIFO channels and *fail-stop*[2] behavior of member processes, this

MP uses a two-phase algorithm for the basic membership update, and a three-phase algorithm when the reconfiguration manager itself fails. This is a centralized protocol, since a single manager process has the responsibility to carry out agreement and commit phases. In the case of the manager's failure, election of a new manager with a consistent membership proposal must avoid *invisible commits*.

The MP presented in this thesis is a decentralized, symmetric protocol, based on a fully connected network of reliable FIFO channels, for asynchronous environments. As in [5], all communication, resulting from communication primitives, between members of a group is assumed to carry a view number. It is required that each increment of the view number be associated with successive views that differ by only one. This protocol guarantees that a given view number is eventually associated with the same membership at all operational members.

The proposed MP eliminates the need for centralizing the responsibility of ensuring consistency of view changes as in [5], by maintaining the group view ordered as a logical ring at each member. Each member perceives the departure (encompassing both failures and voluntary departures) of a neighboring member. Joining members enter on one side of a virtual marker whose position is maintained by all-members.

Reconfiguration of the ring (i.e., updating the membership of the group) is carried out in two rounds, namely *agreement* and *commit*. Agreement and commit actions are achieved using tokens circulated along the logical ring. The protocol is able to regenerate lost tokens and ignore duplicate ones generated during its operation.

The fully-decentralized operation of this protocol sets it apart from all previous work on this field. The concept of a logical ring was also used in [7] and [8]. However, it was used mainly for the purpose of status monitoring. In our case, it constitutes the basis of the entire protocol, and no additional mechanisms are used, in contrast with those protocols.

C. SCOPE AND ORGANIZATION OF THE THESIS

In this thesis, a decentralized asynchronous membership protocol is presented. This protocol was originally presented in [6] and [10]. The present thesis elaborates and presents a revised

version of the basic work, along with a full implementation design, and a discussion of the actual coding process for the FIFO channel Layer interface and all interactions/interfaces with the application.

The thesis is divided in six chapters. Chapter II formally presents the proposed Group Membership Protocol. Chapter III discusses the complete design, covering all processes and interface. In Chapter IV, the implementation details, describing the interface with the operating system and the coding issues, are presented. The performance and scalability of this protocol are analyzed in Chapter V. Chapter VI discusses possible extensions of the protocol and proposes future lines of work in this area. In the Appendix, listings of the developed programs are presented.

II. A DECENTRALIZED GROUP MEMBERSHIP PROTOCOL

In this chapter, a decentralized group membership protocol is described, and proved correct. The original development of this protocol was presented in [6]. This chapter presents the result of successive revisions and refinements motivated by the implementation effort discussed in the next chapter.

A. GROUP MEMBERSHIP PROTOCOL OVERVIEW

1. Assumptions

The proposed Membership Protocol (MP) makes the following assumptions. A fully-connected network (implying no network partition) of reliable (implying that a message is never lost by the network, except in the presence of failure) FIFO (implying that the order of message delivery between a pair of processes is preserved) communication channels connecting operational members is assumed. All failures are assumed to be *crash* or *fail-stop* [1]. This implies that a message sent will not be delivered only because of the receiver's failure. However, it may be arbitrarily delayed. Note that the implementation presented in the following chapter will allow the relaxing of the reliability and FIFO characteristics of the communication channels, but as far as the discussed protocol is concerned those conditions will be assured, by providing the necessary mechanisms within the implementation.

Continuous changes to the membership are allowed; however, the changes are committed one at a time, and with a specific order, common to all members of the group.

A member gets added when a *join* is processed and gets deleted when a *departure* is perceived.

A group name is assumed to be public to those elements that may wish to become members by joining the group. It is assumed that there is a mechanism that allows the prospective member to search the appropriate domain to determine the address of a member currently running on some site. This address is used to send a join request which identifies the sender with an unique address.

The protocol maintains three main data structures at each member, *viz.*, the membership list (group view), status table and token pool (all described in the following sections). Since these are read/written by more than one component of the protocol, it is assumed that mutual exclusion is provided for their access in the protocol implementation.

2. Overview

The proposed MP guarantees that the view changes and relative sequence at each operational member are identical. Using a view number in all group-related communication guarantees that reliable communication primitives can be built.

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

A logical ring has no relation with the physical location of the members. Given such a ring and a direction of traversing (arbitrarily, *clockwise* is selected), each member periodically queries its anti-clockwise neighbor for its status. The neighbor then responds with a status message. It, in its turn, sends a status query to its own anti-clockwise neighbor. Thus, every member monitors one and only one other member and is itself monitored by a third member.

As an example let us consider a group where there are six members p_0 to p_5 , and a logical ring can be configured in such a way that p_0 is the anti-clockwise neighbor of p_1 and clockwise neighbor of p_5 , p_1 is the anti-clockwise neighbor of p_2 and clockwise neighbor of p_0 , and so on. Member p_1 sends a status query to p_0 and p_0 responds with a status report back to p_1 . The numbering of the members p_0 to p_5 is determined by the order of joining to the group, as discussed later on. This process is illustrated in Figure 1.

Initially, the ring configuration is known to all the members (typically the ring will start with one single element, and other elements join in some arbitrary order). As the membership changes, the ring configuration changes. The MP treats the cases of a member leaving the group in the same manner as a member joining the group (failures amount to a member leaving involuntarily).

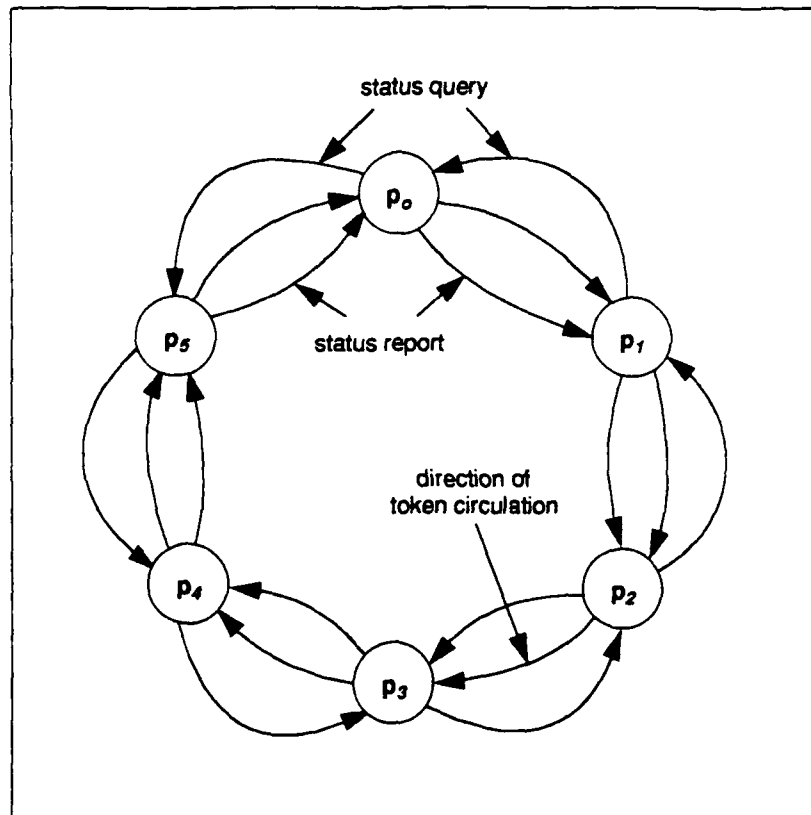


Figure 1 A Logical Ring

and recoveries amount to a member joining as a new one). The protocol maintains appropriate information at operational members to determine whom each member must monitor. When a member departs voluntarily, it simply stops responding to the status query from its monitor. If a failure occurs, it is unable to respond to its monitor, or its response is delayed beyond some reasonable interval of time. In either case, if a monitor does not receive a status message within a certain time interval after sending a query, the monitored member is perceived to have left the group. It is desirable that a member who is alive but has been perceived as departed by its monitor, perceive its own departure from the group. This will allow for the member to gracefully cease its existence, and possibly to later rejoin.

A sequence of actions to ensure that all the operational members consistently commit to this change is then invoked. When a member recovers or wishes to join anew, it sends a join request to the first group member it can locate in the network. This member registers the request and

invokes a sequence of actions, similar to that of departure processing, to ensure that consistent integration of the incoming member takes place.

a. Processing of Individual Changes

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

Departure Processing:

Once a member perceives the departure of its monitored member because it does not receive a status message in response to its query for a predetermined time interval, it initiates the *agreement* phase by sending an agreement token to its clockwise neighbor. It also starts monitoring the anti-clockwise neighbor of the member perceived to have departed.

The agreement token is passed around the ring in the clockwise direction by each member, who passes it along to its own 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 perceived departure. The agreement initiator then starts the *commit* phase by generating a commit token which is circulated along the ring in the same manner as the agreement token. All the members receiving this token commit the change by removing the departed member from their group view and updating the view number.

Join Processing:

The protocol maintains a logical marker in the ring as the position between some pair of adjacent operational members. The clockwise member of this pair is designated as the *host* of the logical ring and is known to all members of the group (it is also the oldest member of the group,

in the sense that it was the first of the current members to join the group). A new member always joins the group as the anti-clockwise neighbor of the host, who has the responsibility of carrying out the agreement and commit phases for the join process. Host status is inherited by the clockwise neighbor of the departing host, thus ensuring that the current host is the oldest member of the group, as described earlier.

Although the host has the responsibility of establishing the join process, the joining member does not necessarily have knowledge of which member is currently the host. A potential member only needs to send a join request to a member it can locate. The member that receives this join request registers it and sends it clockwise along the ring. When the request reaches the host, it takes on the responsibility of carrying out the agreement and commit phases of the join in a manner similar to the departure processing.

The host eventually makes the prospective member its monitored neighbor and delivers it its local membership view, view number, and other related information.

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

3. Definitions

Each member maintains a set containing all the operational members corresponding to its current group view. In addition, each member maintains a status table which stores the perceived state of all the members that are in the process of departing or joining. This table is used by a member to reject any duplicate tokens generated due to the departure of a member in the ring in the middle of any phase.

There is a token 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 used to guarantee that no token is lost upon the failure of a member.

Using the current group view and the local status table, each member determines the member it must monitor.

a. Group Membership Problem

Every member, p_i , associates an integer, vn , with its current group view denoted by the set $GV_{vn}(p_i)$, and increments it by one for every view change committed. Solution of the group membership problem requires that [5]

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

An MP is safe if it guarantees the above. In the following, unless necessitated by the context, the view number will be dropped as a subscript, since it is same across a consistent cut of the group [5].

Note that the above condition holds for only those views that include both elements p_i and p_j .

b. Logical Ring

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

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

c. 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*.

d. 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 $p_i \xrightarrow{p_k} p_j$

For a logical ring, a hypothetical marker fixed along the ring is defined as follows.

e. Logical Marker

A logical marker is a fixed imaginary position between some pair of members along the logical ring.

Its adjacent members may change due to departures and joins.

f. Ring Host

p_{host} is the first operational member clockwise from the logical marker. This member is always the oldest operational member on the group.

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} .

g. Rank

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

This protocol ensures that $rank_{p_i}(p_j) = rank_{p_j}(p_i) \quad \forall p_i, p_j$ and p_j for a given group view. Thus the rank represents global knowledge derived eventually by all the members.

h. Monitoring Member

Every p_i maintains $p_{mon}(i)$ as the last member to have queried it for its health status.

i. Tokens

The proposed MP is based on circulation of three types of tokens to achieve agreement and consistent commit among members. The agreement token initiated at p_i for p_j perceived to have departed or joined is denoted as $agree_{p_i}(p_j)$. Similarly, the commit token initiated

at p_i for p_j perceived to have departed or joined is denoted as $\text{commit}_p(p_j)$. Every token carries information about whether it is for a departure or join.

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

It should be noted that the initiators of the agreement and commit tokens for a given change need not to be identical, and also need not to be the same as the member that perceived the change in the first place. For example, 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 p_2 , and then initiate agreement for p_1 since this member is supposed to be the new anti-clockwise neighbor but it fails to respond to queries. If p_3 fails before the agreement phase is complete, then its neighbor p_4 would assume the responsibility for committing the failure of p_3 , p_2 and p_1 .

j. Local Information

Every member p_i maintains a pool of all the tokens it processes, denoted as $\text{TokenPool}(p_i)$, in the order they are processed. The purpose of maintaining this pool is to store information related to the execution state of the protocol for on-going membership changes. This pool is used to ensure that no tokens are lost due to the departure of the token receiver, before completing the receiving process or immediately after receiving it and before passing it along.

Tokens from this pool are deleted carefully by following the principle that a token is retained at a member until it is guaranteed that its use is complete. The token pool update policy is described later on.

Every member p_i maintains a local status table, denoted as ST_{p_i} . This table maintains information about a member's local view of the extent to which processing for committing membership changes has progressed. A member has an entry in this table at p_j only if it has been

perceived to have departed but not yet committed into $GV(p_i)$, or if it is perceived to have joined but is not yet committed into $GV(p_i)$. This property is crucial to the safety of the protocol.

The five possible values of $ST_{p_i}(p_j)$ are: *Departure Agreed*, *Join Agreed*, *Departure Pending*, *Join Pending* and *Join Requested*.

The *pending* status is used to delay the committing of a change at a particular member so that the order of changes at all operational members is identical. The rank of a member is used to determine if this status should be assigned to a member at the time the commit token for it has been received. Interpretation of the status table entries is summarized in Table 1.

Table 1 INTERPRETATION OF $ST_{p_i}(p_j)$

Departure Agreed	Agreement token for departure of p_j received, but it is not committed; $p_j \in GV_{p_i}$ is true
Join Agreed	same as above for a join; $p_j \notin GV_{p_i}$ is true
Departure Pending	Commit token for departure of p_j received, but it is not processed; $p_j \in GV_{p_i}$ is true
Join Pending	same as above for a join; $p_j \notin GV_{p_i}$ is true
Join Requested	p_i has seen the join request from p_j on its way to the host; $p_j \notin GV_{p_i}$ is true

The status table does not provide any additional information on the group status, when compared with the group view and the token pool. It is totally derived from the local token pool, and constitutes a method of fast look-up that simplifies the implementation and the explanation of the protocol.

k. Neighbor and Host Computation

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

Rule to determine a new p_{host} :

At p_i , $p_{host} = p_j \in GV(p_i)$ such that $\forall p_k (p_k \neq p_j) \in GV(p_i), p_j \xrightarrow{p_{old}} p_k$

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 group views are consistent, this ensures that all the members arrive at the same p_{host} . ST_{p_i} reflects local knowledge and is not necessarily consistent along a cut of the group, so it should not be used in global decisions.

This rule is applied whenever there is a removal of a member committed. It should be noted that the ring relation with respect to p_{old} used here is defined on the current view, because the rule is applied at the time of a commit.

Rule to determine $cwnbr(p_i)$:

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

This rule is applied whenever a 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 $\forall p_k (p_k \neq p_j) \in GV(p_i), p_k \xrightarrow{p_i} p_j$ and $p_j \notin ST_{p_i}$.

This rule is applied to determine the anti-clockwise neighbor. The anti-clockwise neighbor is always obtained from the current group view.

B. THE GROUP MEMBERSHIP PROTOCOL

In Figure 2, the interaction of the MP with the application and the network is shown. The

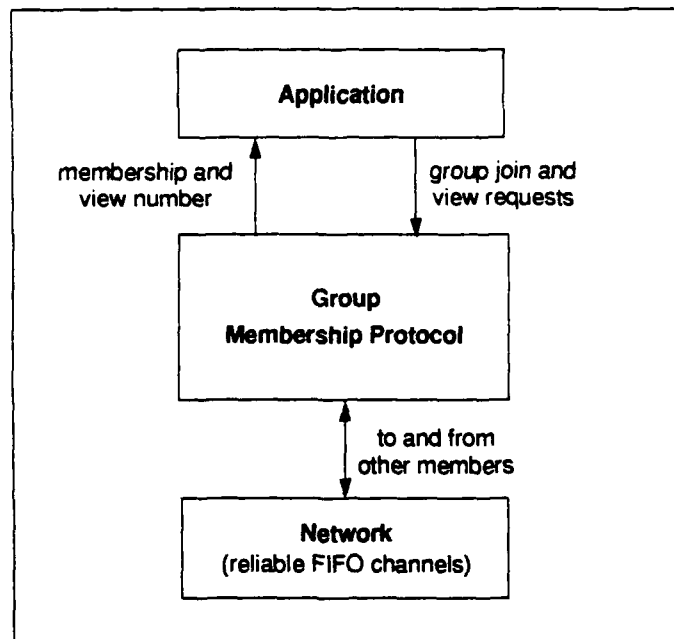


Figure 2 MP Interaction With the External World

network is abstracted as a set of reliable FIFO channels. The application process generates a request to join a particular group and receives the current view of the group.

In the case of a join to an existing group, the MP has the ability to obtain the address of a site (or list of sites) where a member of the requested group is expected to be running. If no site with a running member of that group is found, the MP starts a new group.

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

1. Status Change Detection

Figure 3 shows the algorithm each member executes to monitor its anti-clockwise neighbor and initiate an agreement token if a departure is detected. The Monitor process is

```

Monitor process at  $p_i$ 
1 while (true)
2   send status query to  $acwnbr(p_i)$ 
3   start timer for  $T_{pad}$  /* local timeout interval */
4   wait for incoming message
5   if (timeout)
6     send initiate agreement to ProcessAgreeTkn for the departed member
7     block until ProcessAgreeTkn acknowledges end of processing
8   else if (status report received)
9     wait for  $T_{qry}$ 
10  end
11 end
    end Monitor

```

Figure 3 Monitoring and Agreement Initiation Actions

triggered by the local clock. The clockwise and anti-clockwise neighbors are computed according to the rule given earlier in every iteration of the main 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 message), updates the local status table, generates and adds an agreement token to the local token pool, and sends it to its clockwise neighbor.

If the clockwise neighbor turns out to have already departed, the status reporting instrument shown in Figure 4 ensures that the token will get sent to the next clockwise operational

```

ReportStatus process at  $p_i$ 
1  $p_{mon}$  = querying member
2 send status to  $p_{mon}$ 
3   if (previous querying member  $\neq p_{mon}$ )
4     send  $TokenPool(p_i)$  to  $p_{mon}$ 
5   end
    end ReportStatus

```

Figure 4 Reporting of the Member Status

member. When a change in the querying member is detected, the token pool gets sent to the new querying member, in addition to the status response. The change of the querying member is

recognized by inspecting p_{mon} and comparing it to the originator of the current status query. **ReportStatus** does not compute the clockwise neighbor, but simply responds to the sender of the query (see rule to determine new $cwnbr$, earlier in this chapter).

When the application generates a request to join a group, an instance of the MP gets spawned. It obtains the address of a site running a member, sends a join request message to it, and waits for an intimation of the request approval for a preset interval, before re-sending the request. Before the request is resent, the site is searched again for other members. In the case of no success, other sites are searched in the same manner. This ensures that, if the member receiving the request departs before processing it, the following request will be sent to a different member of the group.

The receiving member p_i runs an algorithm as specified in Figure 5.

```

InitiateJoin for a join request message/token for  $p_{new}$  at  $p_i$ 
1 while (true)
2   if ( $p_{new} \notin ST_{p_i}, GV_{p_i}$ )
3     receive join request message or token for  $p_{new}$ 
4   end
5   if ( $p_i = p_{host}$ )
6     send initiate agreement message to ProcessAgreeTkn for  $p_{new}$ 
7     block until ProcessAgreeTkn acknowledges end of processing
8   else
9      $ST_{p_i}(p_{new}) \leftarrow JoinRequested$ 
10    add  $joinreq_{p_i}(p_{new})$  to  $TokenPool(p_i)$ 
11    if (join request message) /*  $p_{new}$  locates  $p_i$  and sends its join request */
12      generate  $joinreq_{p_i}(p_{new})$  token
13    end
14    send  $joinreq$  token to  $cwnbr(p_i)$ 
15  end
16 end
    end InitiateJoin

```

Figure 5 Processing of a Join Request Message/Token

A non-host member, receiving a request message for the first time, generates the $joinreq_{p_i}(p_{new})$ token and adds it to the local token pool. It enters the *Join Requested* status for p_{new} in its status table and sends the token to its $cwnbr$.

A duplicate join request is rejected if there is an entry for p_{new} in the local status table (i.e., this particular join request was already received and processed).

If the member receiving the request message or the corresponding token is the ring host, it generates the agreement token, updates the local status table and token pool, and sends it to its cwnbr.

2. The Agreement Phase

The algorithm used to process an agreement token is shown in Figure 6.

If the member that receives an agreement token for the first time is not its initiator, it must simply pass it on to its clockwise neighbor, after adding it to its token pool and updating the local status table (lines 10-14). However if it is the initiator of the token, it must generate a *commit* token because the agree token has circulated back to it.

The receiver of a duplicate *agree* token must also generate a *commit* token if the agreement initiator and all members in between had departed after generating the agreement token. This indicates that the token has completely circulated around all the surviving members. In this case, the member generating the commit token will have an entry in its local status table for the initiator of the token, and all members in between (line 8).

Any member commits a change to its view when it processes a commit token for the change. Thus, the initiator of a commit token commits the corresponding change locally and sends it to the clockwise neighbor.

There are two aspects to committing a change in the group view in this protocol. Firstly, since the ring configuration may lead to the arrival orders of two commit tokens to be opposite at two different members, there must be a mechanism to ensure that changes are committed in a consistent order at all members. Secondly, when a change is committed, it must be ensured that all protocol-related entities are correctly updated.

The correct ordering of all changes is based on the rank of the member whose status change is being processed. The ordering is imposed at the initiator of the commit token as follows: if the rank of the member with the changed status is the lowest among all the members for which

```

ProcessAgreeTkn for  $agree_{p_i}(p_k)$  at  $p_i$ 
1 if (initiate agreement message received) /*  $p_i = p_j$  */
2   add  $agree_{p_i}(p_k)$  to  $TokenPool(p_i)$ 
3    $ST_{p_i}(p_k) \leftarrow DepartureAgreed$  or  $JoinAgreed$ 
4   send  $agree_{p_i}(p_k)$  token to  $cwnbr(p_i)$ 
5   send acknowledge to calling process
6 else /* a token is received */
   /* a commit must be generated either when I am the agreement initiator or
   when a duplicated token is received due to departure of the agreement
   initiator  $p_j$  */
7   if ((join &&  $p_k \in ST_{p_i}, GV_{p_i}$ ) || (departure &&  $p_k \in GV_{p_i}$ ))
8     if ( ( $p_i = p_j$ ) || { ( $p_i \neq p_j$ ) && (duplicate token) &&
        ( $\forall p_l$  s.t.  $p_l \rightarrow p_i, p_l \in ST_{p_i}$ ) } )
        /* agreement phase completed, hence initiate commit phase */
9       compute rank  $\forall p_l \in ST_{p_i}$  with Agreed status
10      if (rank( $p_k$ ) is smallest)
11        send an initiate commit message to ProcessCommitTkn
           for this status change
12      else
           /* depending upon whether for join or departure of  $p_k$  */
13         $ST_{p_i}(p_k) \leftarrow DeparturePending$  or  $JoinPending$ 
14      end
15    else
16      if ( ( $p_i \neq p_j$ ) && (not a duplicate token) )
17        add  $agree_{p_i}(p_k)$  to  $TokenPool(p_i)$ 
18         $ST_{p_i}(p_k) \leftarrow DepartureAgreed$  or  $JoinAgreed$ 
19        send  $agree_{p_i}(p_k)$  token to  $cwnbr(p_i)$ 
20      end
21    end
22  end
23 end
end ProcessAgreeTkn

```

Figure 6 Processing of Agreement Tokens

there is an agreement token in the token pool, a commit token is generated. Otherwise, commit token generation is kept pending until all changes for members with a higher rank have been committed (lines 10-14). The eventual removal of the pending tokens is described during the discussion of the commit phase, later on. It is emphasized that a departed member can get a pending status assigned *only at the member that initiates* a commit token for the departure.

To ensure that due to a failure of the anti-clockwise neighbor, agreement and join request tokens are not processed for elements whose transition phase is completed (and as a result the corresponding tokens are no longer in the local token pool), line 7 rejects join request and agree tokens for elements that are been integrated (or have already been integrated) in the group, and rejects fail agree tokens for elements no longer belonging to the group.

3. The Commit Phase

The processing of a commit token, as it circulates around the ring, is shown in Figure 7.

```

ProcessCommit for  $commit_p(p_k)$  at  $p_i$ 
1 if (initiate commit n. message received)
2   generate commit token
3   token to be processed  $\leftarrow$  generated token
4 else if (not ( $p_i = p_j$ )  $\parallel$  (duplicate)))
5   token to be processed  $\leftarrow$  received token
6 else
7   exit
8 end
9 CommitChange
10 while ( $\exists p_l \in ST_{p_i}$  with a higher rank & pending status received
        before  $agree_{p_i}(p_k)$ )
11   CommitChange in rank order
12 end
end ProcessCommitTkn

```

Figure 7 Generate/Receive and Process a Commit Token

If a member is the commit initiator (i.e., the token has circulated back) or if the commit token is received again, it simply exists and no action is taken (line 7).

If the commit token is received for the first time at a member, appropriate commit action must take place (line 9).

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 are now processed (lines 10-12).

The update of all the protocol-related quantities upon committing a change is encapsulated as **CommitChange**, whose steps are shown in Figure 8. These steps are assumed to

```

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

```

Figure 8 Actions for Committing a Change

be executed atomically, so that all status changes are mutually exclusive (serialized and non-reentrant).

Aside from passing the token on to the clockwise neighbor, the local group view, view number and token pool must be updated. Line 4 determines the token pool update policy that garbage-collects old commit tokens. The principle followed in this update is that a token should be deleted from the token pool only when the member is certain that its use is over.

A member keeps its token pool ordered according to their arrival times, inspects all the tokens in it, and deletes all the commit tokens received before the agreement token for the change committed. The commit token just processed is not deleted when the member it is sent to departs before receiving it. This update policy exploits the fact that the group members are connected in a ring built over FIFO channels.

Depending on whether a departure or join is committed, different special actions are required. For example, if the departure being committed is for the ring host, the member determines a new p_{host} (lines 10-12) according to the rule given earlier in this chapter. If the member committing a join is the current group host, it updates the anti-clockwise neighbor to be the new member, and sends the local state to it (lines 13-15).

4. Ensuring An Identical Sequence Of Commits

As members perceive departures/joins around the ring, they initiate agreement phases independently. Therefore, in this protocol, it is possible for multiple agreement phases to proceed simultaneously around the ring, resulting in multiple commit tokens that circulate around the ring at the same time.

Consider any two such status changes. they divide the ring in two pieces. Clearly, the order in which the commit tokens for these changes reach the members in these two pieces will be opposite to each other. An identical commit order is maintained in this situation, as specified by lines 10-14 in Figure 6.

When a commit token is to be generated, it is first checked to see if there are any unprocessed agreement tokens in the token pool. If there are some, commits resulting from these are ordered identically around the ring; otherwise, a commit token is generated and the change committed by the commit process at the request of the agreement process (line 11, Figure 6). 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 tokens (lines 10-11). Agreement tokens for joins in the pool are not considered because members always join with the highest rank.

It should be remembered that the rank of a member is its distance from p_{host} in the clockwise direction. If the rank is not the smallest, the local status is marked as pending (line 13 of Figure 6), and the change is committed and propagated at a later time. thus, use of the rank ensures that the pending status for a change gets marked *only* at the commit initiator. Note that the ranks

are evaluated during the agreement process (Figure 6), so they adapt to changes in the ring configuration.

C. SAFETY AND LIVENESS

Based on the protocol description in the previous section, we prove that the protocol solves the membership problem correctly (safety), and that every status change is eventually committed by all operational members (liveness).

Lemma L1: *Every operational member always receives a token for a change if members update their token pool using CommitChange (Figure 8).*

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 commit phase is preceded by the agreement phase. Obviously, $agree_{p_j}(p_k)$ has circulated completely around the ring. Suppose $\exists$ a $commit_{p_l}(p_m)$ received at p_i before $agree_{p_j}(p_k)$. Thus, in between the arrivals of $commit_{p_l}(p_m)$ and $commit_{p_j}(p_k)$ at p_i , $\exists$ a token, viz. $agree_{p_j}(p_k)$, that has circulated around the ring completely. This implies that, due to the FIFO property of channels, $commit_{p_l}(p_m)$ has circulated around the ring completely also, regardless of the locations of p_i , p_j and p_l around the ring. Thus, $commit_{p_l}(p_m)$ has served its purpose and can be deleted from the *TokenPool* at p_i . Therefore, both, $agree_{p_j}(p_k)$ and $commit_{p_l}(p_m)$ have completed their use and can be deleted (lines 4-9 of Figure 8). By adding $commit_{p_j}(p_k)$ to the *TokenPool* at p_i , the *TokenPool*(p_i) update is complete.

Given this update policy, consider that $cwnbr(p_i)$ fails before receiving the token sent to it. When p_i receives the status query from $cwnbr(cwnbr(p_i))$ due to this failure, *TokenPool*(p_i), that contains the token, is sent to the new clockwise neighbor of p_i by **ReportStatus**. The token pool preserves the FIFO order in which tokens have been received, had there been no failures. Therefore, every operational member always receives a token relative to any change. •

In fact, a string of departures with a length one less than the number of members that have processed a given token can be tolerated before token loss occurs. In this pathological case, the protocol ensures that a legal action is taken. For example, if the join request token gets lost due to a string of departures, the potential member times out and re-initiates its search for another member

of the group it wants to join. In the case of a departure, it is shown later on (Theorem T2) that a departure is always detected eventually and committed by all operational members, regardless of the length of the string of departures.

Lemma L2: *Exactly one p_i determines itself to be p_{host} .*

Proof: **CommitChange** determines a new 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 the new p_{host} . According to Lemma L1, no tokens are lost. Therefore, the commit token for the departure of the old host is processed by every member. Since the host had $rank = 0$, which is always the lowest, every member determines the same member as the new p_{host} . •

Lemma L3: *A newly joined member receives all the tokens for uncommitted changes and a consistent group view.*

Proof: p_{host} sends its GV , vn , ST and $TokenPool$ to the joining element p_{new} . The exception to the rule to compute $acwnbr$ ensures that the logical ring is correctly configured with p_{new} as the highest ranked member. When the old $acwnbr(p_{host})$ notices that the querying member is different from its p_{mon} , it becomes aware of the new member in the ring and sends its $TokenPool$ 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 T1 (Safety): *The proposed protocol correctly solves the GMP stated as*

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

given that all members start with the same initial group view, i.e.,

$$GV_0(p_i) = GV_0(p_j), \forall p_i, p_j \in GV_0(p_j)$$

Proof: We provide a proof by induction.

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

Induction Hypothesis: Assume that:

$\exists k > 1 \in N$ such that $\forall p_i, p_j \in GV_k(p_j), GV_k(p_i) = GV_k(p_j)$

We now prove that the next change committed by any two members that continue to remain operation is identical. Consider any $\forall p_i, p_j \in GV_{k+1}(p_j)$. Without loss of generality, let $commit_{pk}(p_i)$ be the next change to be committed by member p_j .

Without loss of generality, assume that $p_j \xrightarrow{p_k} p_i$. It is clear from the change detection instruments that $p_j \xrightarrow{p_k} p_i$ and $p_i \xrightarrow{p_k} p_i$. Therefore, if a change involving p_i is view change (k+1), committed at p_j , either the only agreement token p_k has at the time of initiating $commit_{pk}(p_i)$ is for p_i , or p_i has the smallest rank among all agreement tokens in the *TokenPool* at p_k . Now, a commit token initiated for p_m , such that $p_m \xrightarrow{p_j} p_i$ cannot result in view change (k+1) at p_i , because this implies that p_m has a lower rank at p_i than p_i , whose agreement token will be part of the *TokenPool* at p_i . Therefore, agreement token for p_m would also be present in the *TokenPool* at p_k and would have the smallest rank at the time of initiation of $commit_{pk}(p_i)$. This contradicts the fact that p_i 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 $commit_{pk}(p_i)$.

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.

Theorem T2 (Liveliness): *For any change, the agreement phase is always started.*

Proof: In the case of a departure, the member who perceived it may itself depart before initiating the agreement token, or after sending it. In the latter case, the agreement phase that has been started will result in circulating the agreement token back to $cwnbr(p_i)$ sometime. the commit phase is then carried out by $cwnbr(p_i)$ (line 8 of Figure 6). In the former case, $cwnbr(p_i)$ perceives the departure of p_i and initiates an agreement phase. It attempts to monitor $acwnbr(p_i)$, whose agreement p_i could not initiate. $cwnbr(p_i)$ perceives $acwnbr(p_i)$ as departed also and initiates an agreement phase for it. This sequence of events is extended if there is a string of departures. Therefore, the agreement phase for a departure is always started.

In case of a join, if p_i is the host and fails before initiating the agreement phase for a join, $cwnbr(p_i)$ determines itself to be the new host and receives the *joinreq* token as part of the TokenPool from its new $acwnbr(p_i)$. it can now initiate the agreement phase. Since every operational member always receives a token (by L1), once a join request has been received and propagated by an operational member, an agreement phase for its join is always started.

In the pathological case, when the member that receives the join request message departs before propagating it, or when there is a string of failures that result in a total loss of the join request, the joining member will timeout waiting for the process to complete, and it will restart the joining procedure. •

III. PROTOCOL DESIGN

In this chapter we discuss the implementation aspects of the proposed Group Membership Protocol. General design principles are presented first, and a detailed break-down of the protocol is performed by specifying the individual processes.

The protocol runs as a distributed script, with multiple instances of "members" residing in one of many sites. Each member is a collection of communicating processes. A member, as an entity, communicates with application processes (delivering the current group view), and with other elements (to carry out the MP script).

The major functionalities of the MP are detection of departure, agreement on departure, committing of departure, addition of members and providing the current group view to application processes.

A. PROTOCOL SOFTWARE DESIGN

The top level interface of the MP element is presented in Figure 9.

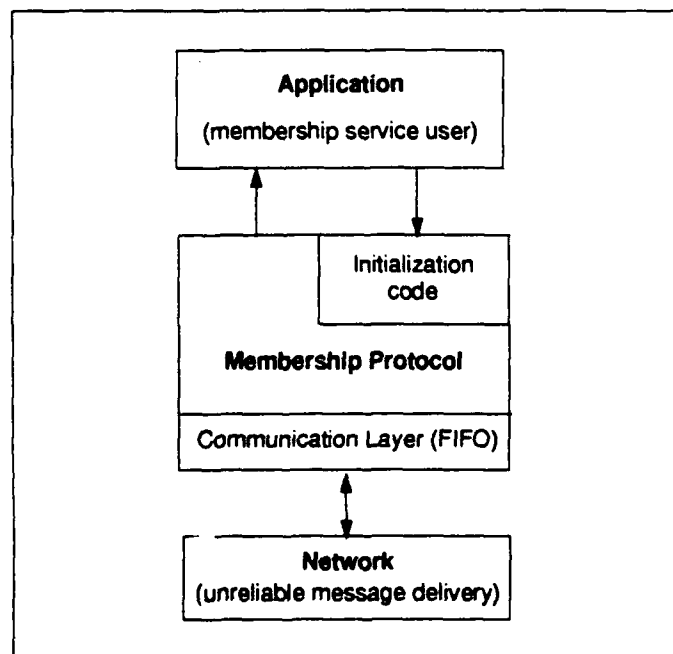


Figure 9 Membership Protocol Interfaces

The ring topology ensures that communication between members is ordered and restricted. Each member has communication links with two adjacent members, namely the clockwise and anti-clockwise neighbors. The exception to this rule occurs when a potential new member joins the group. In this situation a join request message is sent by the potential member to an arbitrary member of the group, and later on, the host sends an initial parameters message to the potential member, to give it member status.

B. PROCESS SPECIFICATION

Each protocol instance is formed by a number of processes, linked by a set of communication channels. The individual processes that form the protocol instance are described in detail in the following sections. Table 2 summarizes all processes used to implement the protocol.

Table 2 PROTOCOL PROCESSES

PROCESS	SUB-PROCESS
Fifo Channel Layer	Front Port
	Back Port
Monitor Process	Status Monitor
	Status Reporter
	Timer
Join Processor	
Agreement Processor	
Commit Processor	
Group View Manager	
Status Table Manager	
Token Pool Manager	

The internal functionality of each process, along with its interactions with other processes, are defined.

1. **Fifo Channel Layer**

This process is responsible for all communications with other elements. It provides the interface to all internal processes needed to propagate or accept messages to/from the network. It is functionally sub-divided in two independent processes: FRONT and BACK.

The FRONT process handles all communications with the clockwise neighbor and the BACK process communicates only with the current anti-clockwise neighbor. The exception to these rules are related to the potential member's join process (see Figure 10). FRONT processes only connect to BACK processes and vice-versa, resulting on a functional decoupling of the two sub-processes.

This decoupling of the two sub-processes allows the element to live as a singular group, thus facilitating the creation of a new group. The process of joining an existing group starts with the creation of a single-element group that later joins the desired group.

In the case of a network that does not provide reliable connections, or when the use of this type of connection is not desired due to other considerations (see discussion on the Unix-based implementation later on), the FIFO-channel-layer provides a form of data link protocol based on a modified *alternating bit protocol* [19]. This protocol uses a serial number that increases by one for each message processed. Like the alternating bit protocol, a message can only be transmitted, after the previous one is positively acknowledged by the destination.

a. Front Sub-process

This sub-process handles the communication with the clockwise neighbor. It also receives Join_Request messages from new members. When joining a group, it receives the Initial_Parameters message from the host.

Front is assumed to be a dual-ported process: it has an internal communication port (at the top of the box in Figure 10), and an external communication port (at the bottom in Figure 10).

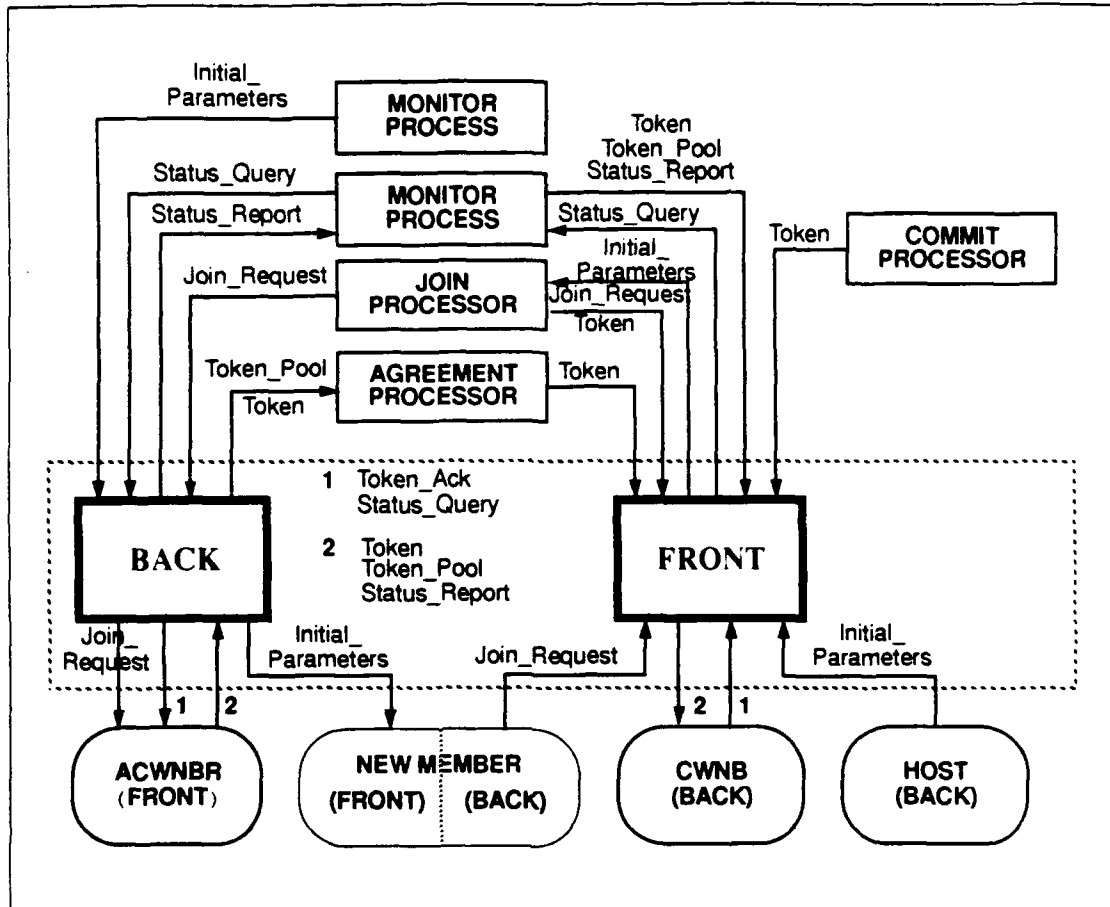


Figure 10 Fifo Channel - Process Dependencies

The clockwise neighbor (cwnbr) is determined by inspecting the destination of the outgoing Status_Report message. It is updated on every status report sent out, thus permitting an asynchronous change in the clockwise neighbor.

Figure 11 describes the full algorithm used to implement this sub-process. In these algorithmic representations, italics are used to denote messages and bold face to represent internal variables, or data structures.

This process executes an infinite loop, checking for the arrival of a message at either of its ports (line 1). Depending on the port that delivers the message, appropriate action is taken.

If a message is received at the external port, it is simply dispatched to the appropriate destination, after being striped from its external header (lines 3-8). Lines 9-11 are used to implement the acknowledgment protocol in the case of a non-reliable network.

```

1 Wait for a channel ready to read
2 if (external channel ready)
3   if (Status_Query)
4     send Status_Query to MONITOR_PROCESS
5   else if (Join_Request)
6     send Join_Request to JOIN_PROCESSOR
7   else if (Initial_Parameters)
8     send Initial_Parameters to JOIN_PROCESSOR
9   else if (Token_Ack)
10    if (Received_Serial_Number = Expected_serial_number)
11      remove Head_of_Queue
12      decrement Queue_Counter
13  end
14 else /*internal channel ready*/
15   if (Token)
16     change Token to external format /* add external header */
17     insert Token in queue
18     increment Serial_Number
19     increment Queue_Counter
20   else if (Token_Pool)
21     discard all messages in queue
22     change Token_Pool to external format /* add external header */
23     insert Token_Pool in queue
24     increment Serial_Number
25     increment Queue_Counter
26   else if (Status_Report)
27     update cwnbr
28     send Status_Report to cwnbr
29   end
30   if Queue_Counter > 0
31     send Head_of_Queue to cwnbr
32     set Expected_serial_number = Head_of_Queue_serial_number
33   end

```

Figure 11 Fifo Channel - Front Process

If the received message comes from the internal port, different actions are taken according to the particular message received.

If a *token* is received, the corresponding external message is assembled (it includes the serial number and the originator address), and inserted in the message queue (lines 16-17). This queue stores all outgoing *tokens* until their reception is positively acknowledged by the destination.

The serial number is incremented, to be used by the next *token* and the counter *Queue_Counter* is incremented to reflect the new queue size. Messages left in the queue are the ones whose transmission is not completed (lines 16-25).

The *Token_Pool* message always follows a *Status_Report* that is sent to a new *cwnbr* (see Monitor Process specification). This means that all messages left in the queue are obsolete and can be discarded (line 21), because all such messages are *Tokens* that are a present on the *Token_Pool*. At this point the message is handled exactly the same way as if it were a *token* (lines 23-25).

When the message is a *Status_Report*, the destination field is used to update the *cwnbr*, and then the message is transmitted. There is no need to apply the acknowledgment protocol here since these messages are not critical to the protocol integrity (if this message is lost, it means that the network is experiencing perturbations, and that will result on a failure assessment by the current monitoring element). This sequence of actions is described in lines 27-28.

Regardless of the message received, if there is at least one outstanding message in the queue (i.e. *Queue_Counter* is non-zero), the message at the top of the queue is sent out to the current *cwnbr*, and its serial number becomes the expected serial number for the next acknowledge (lines 30-32).

Input data flows (internal port): *Token*, *Token_Pool* and *Status_Report*.

Input data flows (external port): *Initial_parameters*, *Status_Query*, *Join_Request* and *Token_Ack*.

Output data flows (internal port): *Initial_Parameters*, *Status_Query* and *Join_Request*.

Output data flows (external port): *Token*, *Token_Pool* and *Status_Report*.

b. Back Sub-process

This sub-process handles the communication with the anti-clockwise neighbor. It also sends *Join_Request* messages from a potential member to an existing member of the group.

When the current element is the host of the group it sends the *Initial_Parameters* message to the joining member.

Like Front, Back is assumed to be a dual-ported process: it has an internal communication port (at the top of the box in Figure 10), and an external communication port (at the bottom in Figure 10)

The anti-clockwise neighbor (acwnbr) is determined by inspecting the destination of the *Status_Query* message.

Figure 12 describes the full algorithm used to implement this sub-process. The process executes an infinite loop, checking for the arrival of a message at either of its ports (line 1). Depending on the port that delivers the message, appropriate action is taken.

If a message is received at the internal port, it has to be transmitted to a destination specified in the message itself. If it is a *Status_Query*, its destination is used to determine the current acwnbr (lines 3-5). In the case of *Initial_Parameters* and *Join_Request*, the messages are simply sent to the designated destination (lines 6-9).

If the received message comes from the external port, it is processed only if the originator is the current acwnbr (line 12). The *Status_Report* message is sent to the Monitor Process. *Token* messages are only accepted if they have the correct serial number (lines 15-20). *Token_Pool* messages are always accepted and set a new sequence for the expected serial number (lines 21-25). Note that *Token_Pool* messages meet the criteria in line 12, because they result from a group reconfiguration triggered by a *Status_Query* message sent to the new acwnbr, that results in the update of the acwnbr (line 4).

Input data flows (internal port): *Initial_Parameters*, *Status_Query* and *Join_Request*.

Input data flows (external port): *Token*, *Token_Pool* and *Status_Report*.

Output data flows (internal port): *Token*, *Token_Pool* and *Status_Report*.

Output data flows (external port): *Initial_parameters*, *Status_Query*, *Join_Request* and *Token_Ack*.

```

1 Wait for a channel ready to read
2 if (internal channel ready)
3   if (Status_Query)
4     update acwnbr
5     send Status_Query
6   else if (Initial_Parameters)
7     send Initial_Parameters
8   else if (Join_Request)
9     send Join_Request
10  end
11 else /*external channel ready*/
12   if (message originator = acwnbr)
13     if (Status_Report)
14       send Status_Report to MONITOR_PROCESS
15     else if (Token)
16       if (Serial_Number = Expected_Serial_Number)
17         send Token to AGREEMENT_PROCESSOR
18         send Token_Ack /*to acwnbr*/
19         increment Expected_Serial_Number
20       end /*out of order messages are discarded*/
21     else if (Token_Pool) /*Token_Pool is always accepted*/
22       send Token_Pool to AGREEMENT_PROCESSOR
23       send Token_Ack /*to acwnbr*/
24       set Expected_Serial_Number = Serial_Number + 1
25     end
26   end
27 end /*only messages from acwnbr are accepted*/
28 end

```

Figure 12 Fifo Channel - Back Process

2. Monitor Process

This process is responsible for checking the health status of the current anti-clockwise neighbor. It does so by sending periodic *Status_Query* messages and checking for the arrival of the corresponding *Status_Report* within a specific time interval. It is also the process that performs the complementary function by sending a *Status_Report* in response of an incoming *Status_Query* from its cwnbr.

The two functions are totally independent, so they are best implemented by two dedicated sub-processes: Status Monitor and Status Reporter.

The maximum delay allowed for a *Status_Report* is determined by another sub-process, Timer. This value can be fixed, or it can be made adaptive to the network conditions.

Figure 13 shows the internal constitution of the Monitor Process and all interdependencies, both within the process and with other processes.

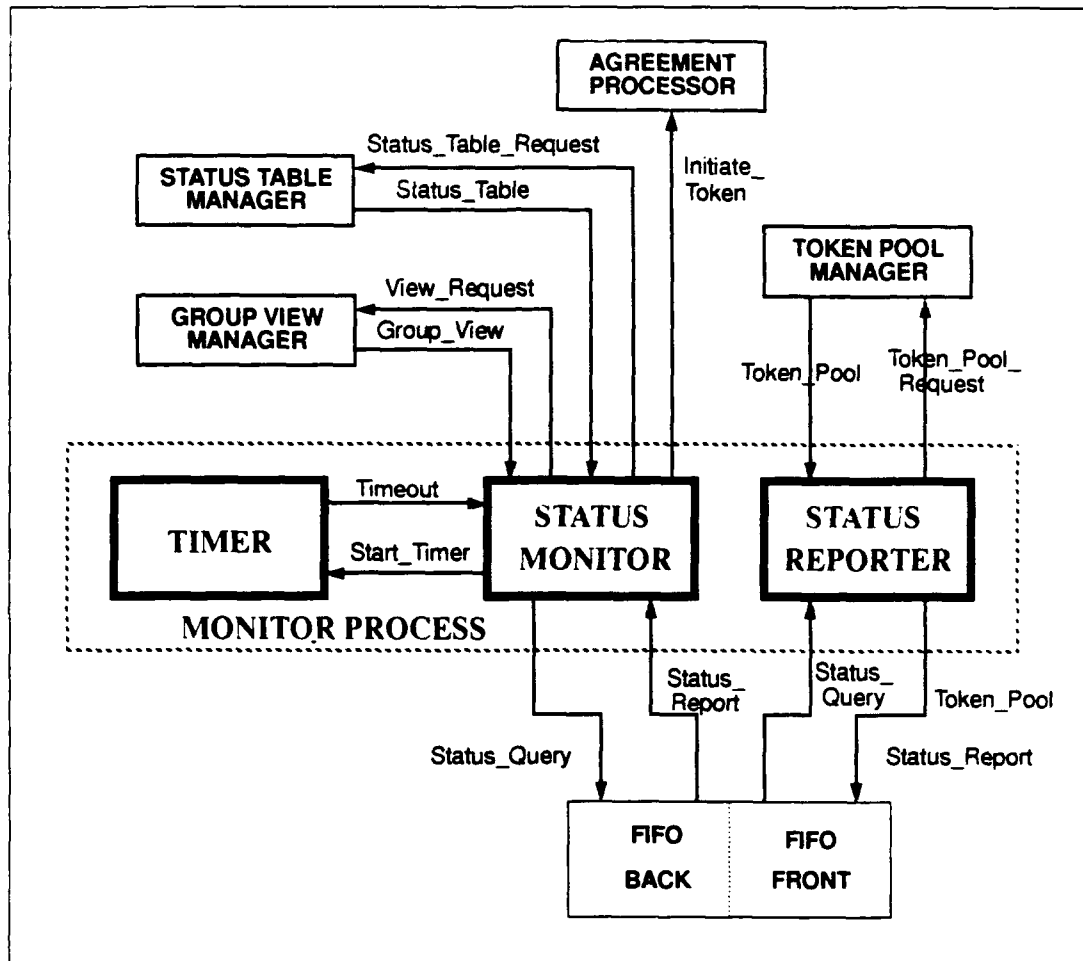


Figure 13 Monitor Process - Internal Structure and Dependencies

Given that this process centralizes the periodic status assessment of the neighbor elements, it could also determine if the element is fully operational before answering to a *Status_Query* request. This option is not used in the current implementation. We rather ensure that an eventual failure in the element will bring all processes down.

a. Status Monitor

This sub-process checks the health of the current anti-clockwise neighbor by periodically sending a *Status_Query* message. It expects a *Status_Report* in response within a time interval defined by the auxiliary sub-process Timer. If a timeout occurs, the neighbor element is assessed as failed and the following actions are taken:

1. An *Initiate_Token* message is sent to the Agreement Process. This starts the agreement phase of the element departure.
2. It determines the new anti-clockwise neighbor based on the current Group View and Status Table, using the procedure described in the previous chapter.
3. Uses the new acwnbr as a target for all subsequent *Status_Query* messages.

It is important that these actions are taken in this strict order. Furthermore, we have to ensure that the Agreement Process completes its processing before we carry on with steps two and three.

Input data flows: *Status_Report*, *Group_View* and *Status_Table*.

Output data flows: *Status_Query*, *View_Request* and *Status_Table_Request*.

b. Status Reporter

This sub-process determines the health of the element and answers to *Status_Query* messages from the current cwnbr.

It keeps track of the element that is performing the queries. If it detects a change of the cwnbr, it sends a *Status_Report* to the new querying element. It then requests a *Token_Pool* from the Token Pool Manager and sends it down to the Fifo Channel Layer, to be sent to the new cwnbr. Note that The Fifo Channel Layer uses the *Status_Report* message to determine the current cwnbr.

Input data flows: *Status_Query* and *Token_Pool*.

Output data flows: *Status_Report*, *Token_Pool* and *Token_Pool_Request*.

c. Timer

This sub-process works as an interruptible stop-watch. It starts a count-down upon receiving a *Start_Timer* message. The count-down can have one of two durations: a query timeout interval, or a query period interval.

The query timeout interval is used to determine the failure of the acwnbr. This interval must be set long enough to allow for reasonable delays in the transmission of a message (token or token pool), its processing at the Fifo Channel Layer of the destination, and the transmission of the corresponding acknowledge message. If the interval is too small, the probability of a wrong assessment of failure increases. An interval too long will cause the protocol to react slower to real failures. It is possible (and probably desirable in some network configurations) to make this interval adaptive to the network conditions.

The query period interval sets a querying period, that determines how often the querying process is to be performed. Using a longer period will reduce the traffic on the network, while a shorter period will improve the failure detection time.

Input data flows: *Start_Timer*.

Output data flows: *Timeout*.

3. Join Processor

This process centralizes the actions during the joining of a new member. Depending on the role of the current element, different sets of actions are taken. These roles are: prospective new member, host of the group, any other element of the group.

The interactions of Join Processor with other processes are shown on Figure 14.

The actions taken by this process depend on the role assumed by the element. They are described below.

When the current element is a **prospective new member** in the process of joining an existing group, or of establishing himself as a new group, the following actions are taken.

The process starts a search for elements of an existing group in a well-known location (i.e., in a specific file at a given set of hosts).

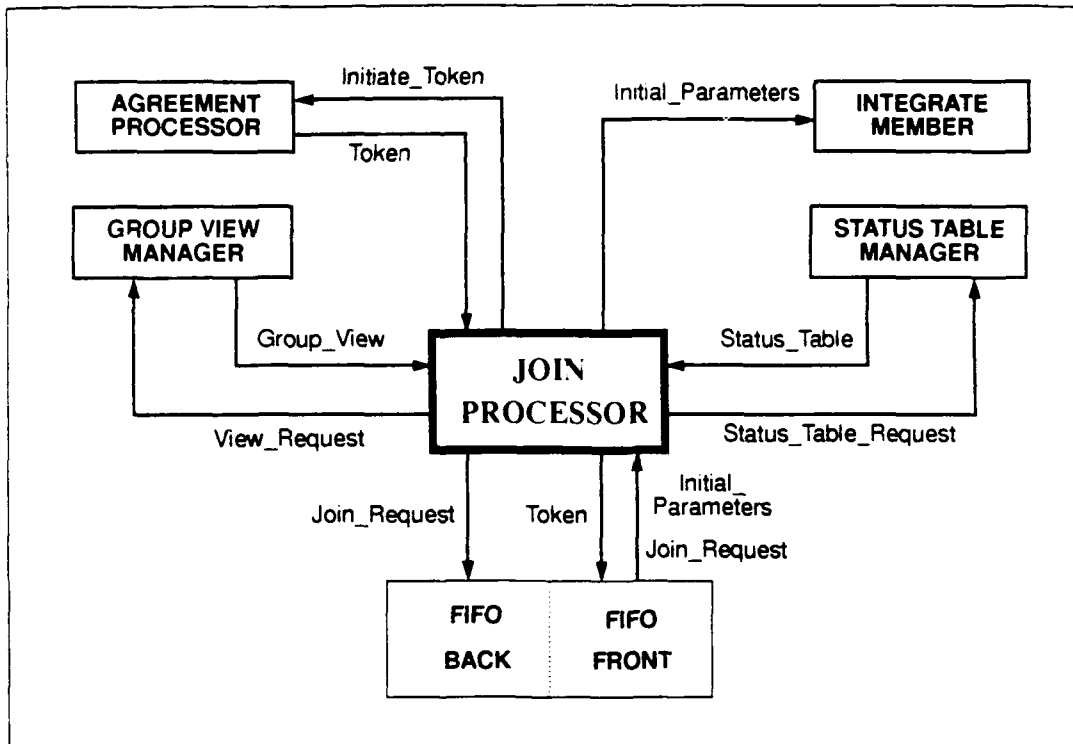


Figure 14 Join Processor - Process Dependencies

If the desired group is not available (because all elements have failed or this is the first element of a new group), then the element is to become the only member of a new group. This is already the case, because the element is initiated such that it becomes an autonomous singular group. The only action to be taken in this case is to create the well known file corresponding to the new group.

If an element is found, the Join Processor sends it a *Join_Request* message and waits for the arrival of an *Initial_Parameters* message. If this message does not arrive within a specific time interval, it is assumed that the recipient of the *Join_Request* could not forward the request. In this case the joining process is repeated, starting with a new search for a living element.

If the *Initial_Parameters* message is received in due time, the Join Processor forwards it to the Integrate Member process.

When the current element is the **host of the group**, the following actions are taken.

If the process receives a *Join_Request* (from the prospective new member), or a *Token* of type *join_request* it as to generate a *Token* of type *join_agree*. Since token generation is centralized at the Agreement Processor, an *Initiate-Token* message is sent to it, requesting the generation and propagation of the *Token*.

When the current element is a **member other than the host of the group**, the following actions are taken.

If the process receives a *Join_Request*, it as to generate a *Token* of type *join_request* (note that this is a different message than *Join_Request*). For this effect an *Initiate-Token* message is sent to the Agreement Processor.

When the process receives a *Token* of type *join_request*, the same token is sent down to the next element.

Input data flows: *Join_Request*, *Initial_Parameters*, *Status_Table*, *Token* and *Group_View*.

Output data flows: *Join_Request*, *Token*, *Initial_Parameters*, *Status_Table_Request*, *Initiate-Token* and *View_Request*.

4. Integrate Member

This process initializes the internal status of the element, such that it becomes consistent with the group. It also provides the complete status to new members joining the group.

Figure 15 shows all the connections of this process.

This process receives the *Initial_Parameters* message, and extracts from it the messages *Initial_Group_View*, *Initial_Status_Table* and *Initial-Token_Pool*. These messages are then sent to the appropriate Manager process.

When the current member is the group host, and the joining process has completed the agreement phase, a *Send_Initial_Parameters* message is received from the Commit Processor. In this case the *Initial_Parameters* message is assembled and sent to the new member.

Input data flows: *Initial_Parameters*, *Send_Initial_Parameters*, *Group_View*, *Status_Table* and *Token_Pool*.

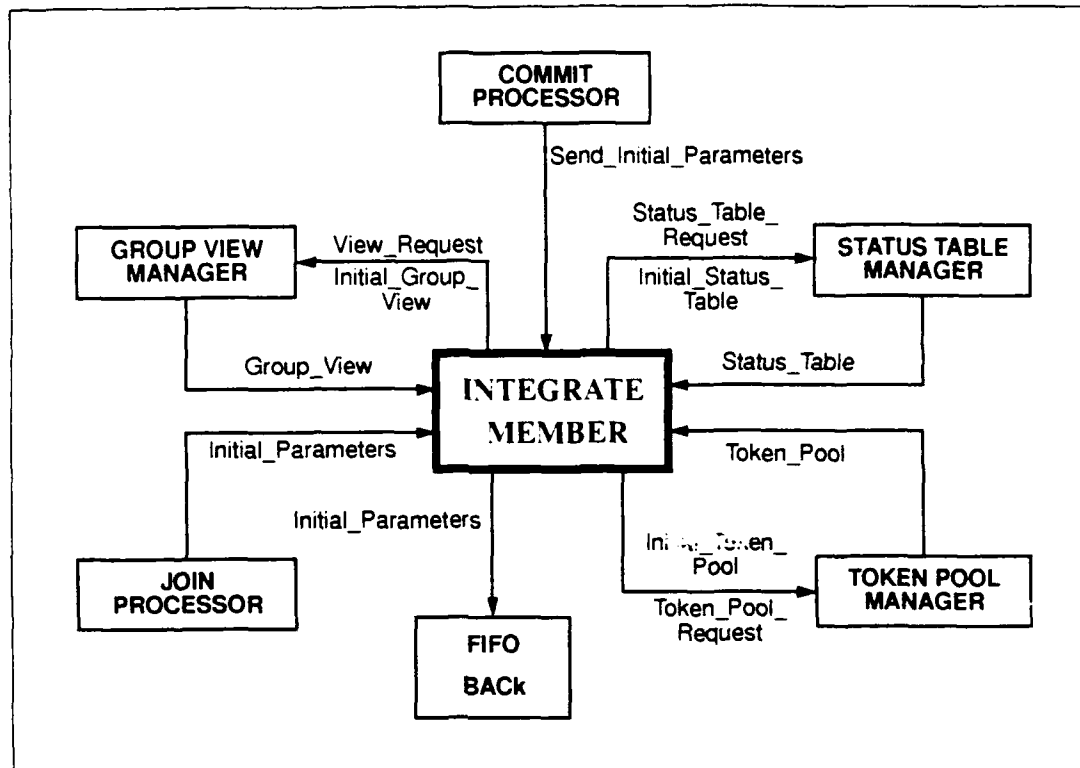


Figure 15 Integrate Member - Process Dependencies

Output data flows: *Initial-Token_Pool, Initial_Status_Table, Initial_Group_View, Initial_Parameters, Token_Pool_Request, Status_Table_Request and View_Request.*

5. Agreement Processor

This process handles all *Tokens* received from the exterior. It receives and processes agreement tokens from other elements, and sends them down to the next member if necessary.

The agreement processor also assembles tokens that are to be locally originated, upon reception of the *Initiate-Token* message, adds them to the Token Pool, and sends them down to the next member.

The above operations implement the processing of agreement tokens described in Chapter II, Figure 6.

All non-agree tokens are dispatched to the appropriate internal process.

Figure 16 details all the dependencies of this process.

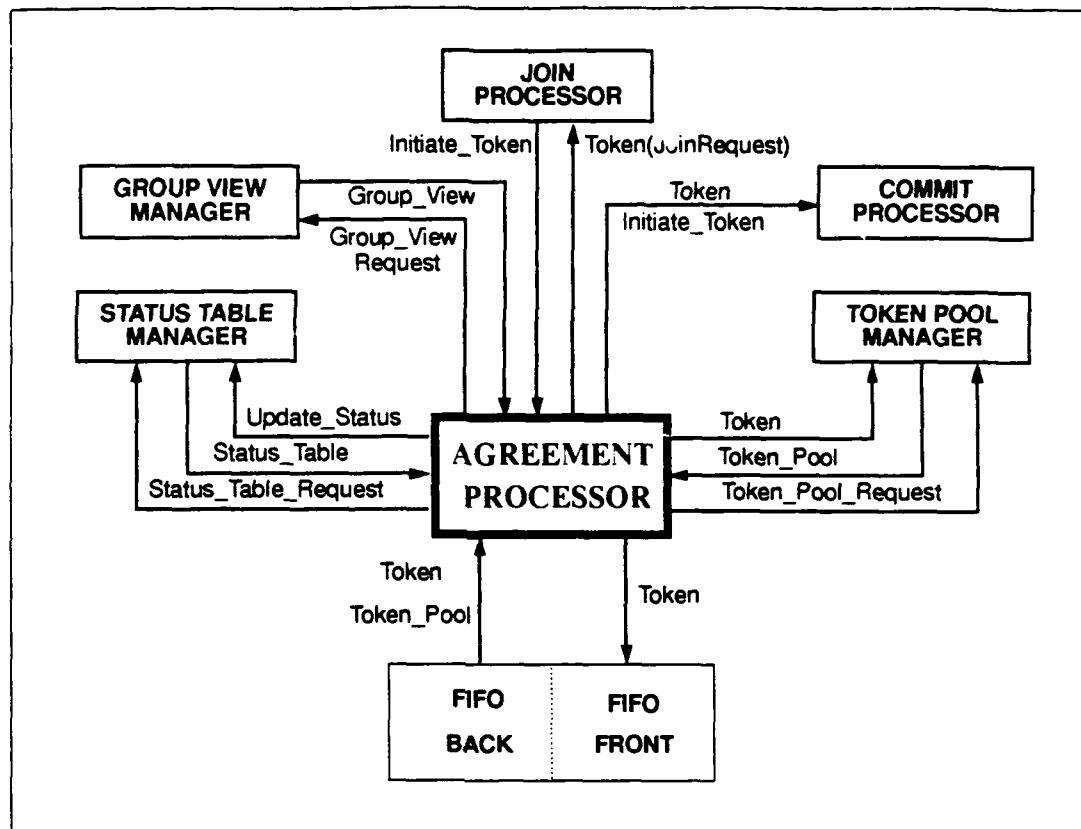


Figure 16 Agreement Processor - Process Dependencies

If the message is an agree token, it is processed and the internal status of the element is updated by sending an *Update_Status* message to the Status Table Manager.

If an agree token is found to have completed its cycle, the commit phase is initiated by sending the *Initiate_Token* message to the Commit Processor. The current *Group_View*, *Token_Pool* and *Status_Table* are used to determine this event, as described in Chapter II, Figure 6.

If the incoming message is a commit, it is sent to the Commit Processor for further processing.

If the incoming message is a join_request token, it is added to the *Token_Pool* and *Status_Table* and then sent to the Join Processor for further processing.

To avoid duplication of the joining process for a member that has just completed the joining process, only join requests and join agree tokens belonging to elements not present in the

local status table and group view are accepted. In the same way, agreements for departure are accepted only if the subject element is present in the group view (see line 7 in Figure 6).

When receiving a token pool, all tokens are extracted and processed if they are not present in the local token pool.

It is important to mention that when searching for a particular token, only the type and the subject are relevant. For example in the token $joinagree_{p_k}(p_i)$, p_i is the subject and $joinagree$ is the token type, while p_k is the token originator. The member responsible for a particular group reconfiguration can change during the process, due to departure of the originator, and so join requests, agreement and commit tokens, referring to the same action, can have different originators.

Input data flows: *Token*, *Group_View*, *Status_Table* and *Token_Pool*.

Output data flows: *Token*, *Initiate-Token*, *View_Request*, *Token_Pool_Request*, *Update_Status* and *Status_Table_Request*.

6. Commit Processor

The commit process is responsible for committing the removal or the joining of an element from/to the group, as described in Chapter II, Figure 7 and Figure 8.

Figure 17 shows the interconnections to other processes.

The inputs to this process are *Token* (commit) and *Initiate-Token*, and they both come from the Agreement Processor.

If an *Initiate-Token* message is received, the corresponding commit *Token* is generated and processed exactly as if it were a token coming from another member.

A commit token is processed if it has not yet been received (i.e. it is not a duplicate). The Group View and Status Table are updated. All commits left pending, referring to an element with higher rank than the current token subject, and received before the agree are now committed. All commit tokens received before the agree token for the element being committed, are deleted from the Token Pool. If committing a join, the corresponding join_request token is also deleted. The commit token is then added to the Token Pool and the corresponding agree is deleted.

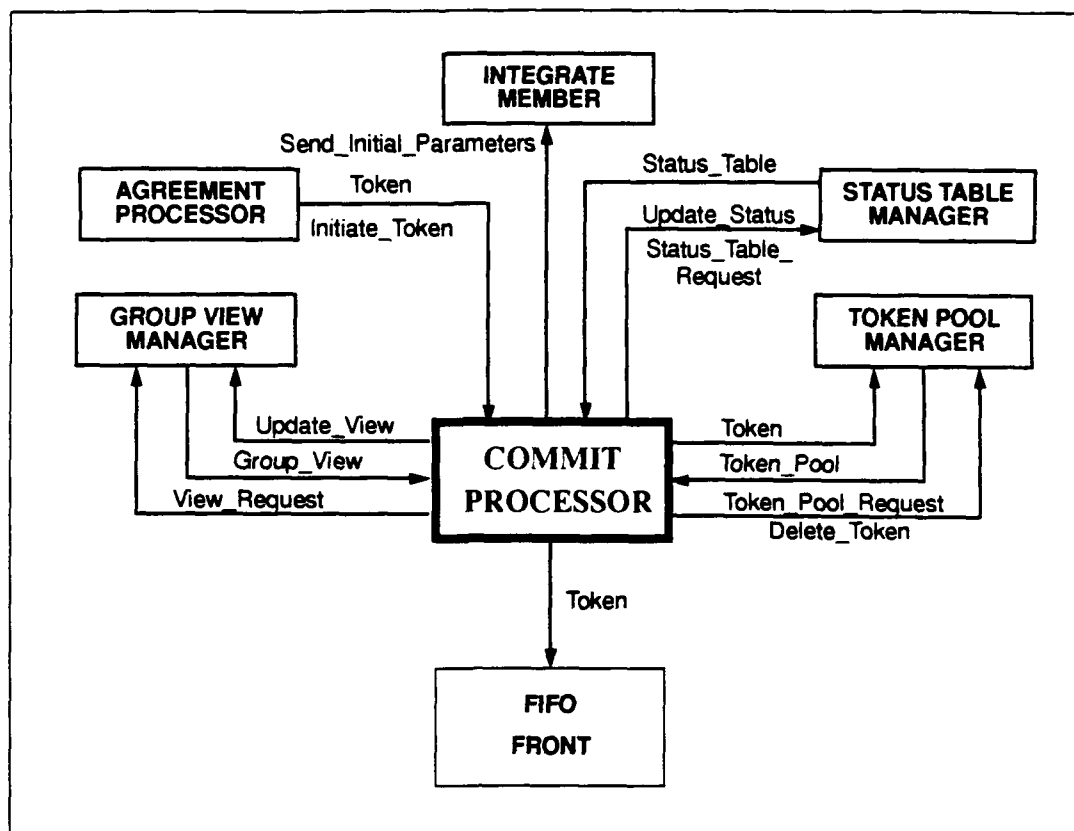


Figure 17 Commit Processor - Process Dependencies

When the current element is the host, and committing a join, a *Send_Initial_Parameters* message is sent to the Integrate Member process.

The current commit token is sent to the next member of the group.

The process of committing changes to the internal status has to run atomically, to ensure a correct manipulation of the Group View (Figure 8).

The same note on token identification previously stated for the Agreement Processor also applies.

Input data flows: *Token_Pool*, *Group_View*, *Status_Table*, *Initiate_Token* and *Token*.

Output data flows: *Token_Pool_Request*, *View_Request*, *Delete_Token*, *Update_View*, *Token*, *Status_Table_Request* and *Send_Initial_Parameters*.

7. Group View Manager

This process manages the membership list (group view) and the view number. The view number is a monotonically increasing value that is set at start-up time or by the *Initial_Group_View* message.

The process dependencies are shown on Figure 18.

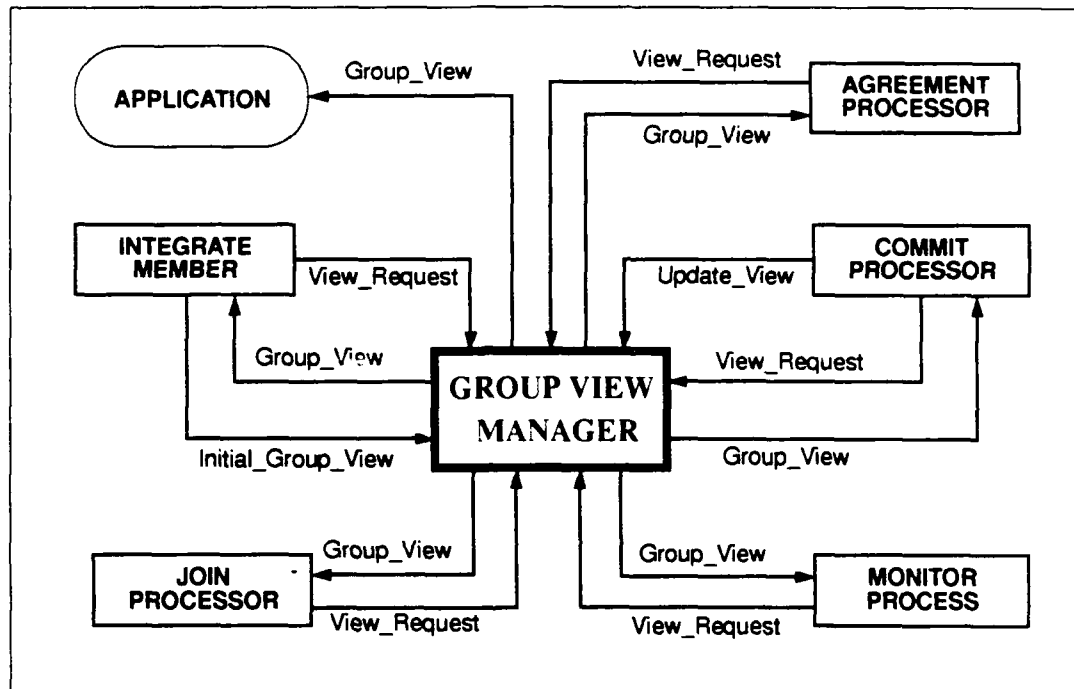


Figure 18 Group View Manager- Process Dependencies

When this process is initialized, i.e. executed for the first time, it has to establish a view that reflects a singular group, with this element as the only member.

When an *Initial_Group_View* is received, the original view has to be discarded and the new view becomes active. This happens as a consequence of a successful join to an existing group.

In response to a *View_Request*, a *Group_View* message is assembled and sent to the client process.

The *Update_View* message has two possible types, add and delete. When an element is to be added to the view, it is always inserted with the highest rank, i.e. anti clockwise from the host

as seen in the logical ring. When an element is to be deleted, the ring is reconfigured such that the rank ordering is maintained.

Every time the Group View is changed, this process notifies the application by sending a *Group_View* message. It also updates the well known file that stores all current members.

Input data flows: *View_Request*, *Update_View* and *Initial_Group_View*.

Output data flows: *Group_View*.

8. Status Table Manager

This process keeps track of the status of all members that are in a transition phase.

The process dependencies are shown on Figure 19.

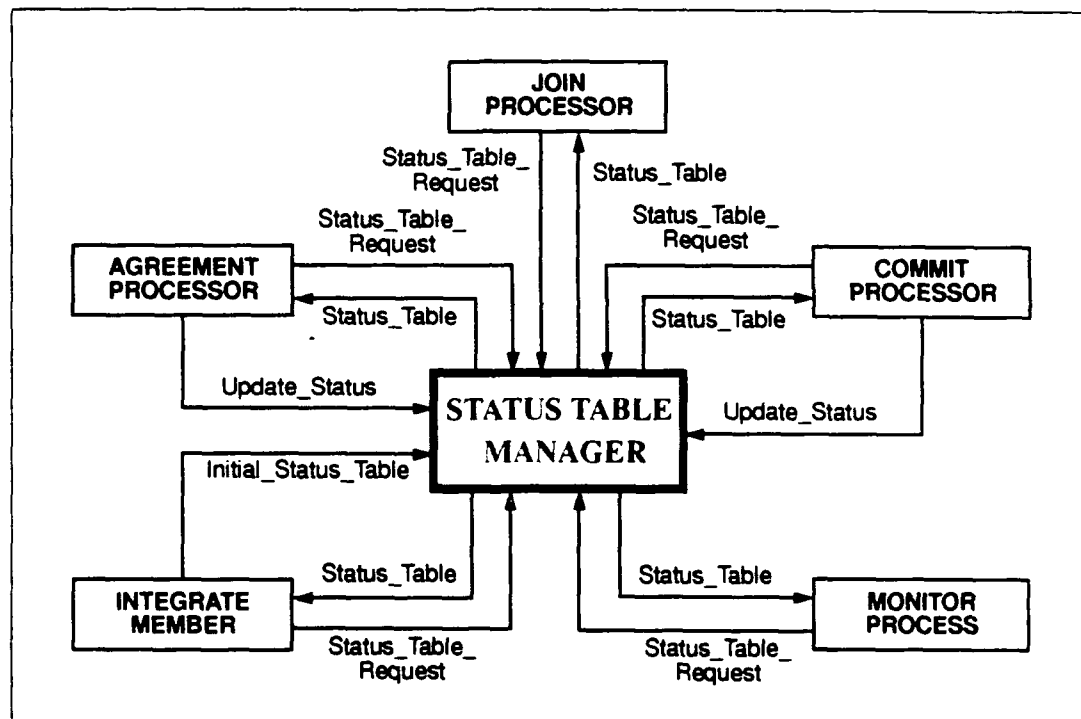


Figure 19 Status Table Manager- Process Dependencies

When this process is initialized, i.e. executed for the first time, it has to establish an empty Status Table.

When an *Initial_Status_Table* is received, it is established as the current one and made active. This happens as a consequence of a successful join to an existing group.

In response to a *Status_Table_Request*, a *Status_Table* message is assembled and sent to the client process.

The *Update_Status* message has several possible types, corresponding to the various transition status specified by the protocol. When the element referenced in *Update_Status* already has an entry on the table, the manager deletes the old entry and inserts a new one. If the element does not have an entry in the table, then a new entry is inserted with the appropriate status.

Input data flows: *Status_Table_Request*, *Update_Status* and *Initial_Status_Table*.

Output data flows: *Status_Table*.

9. Token Pool Manager

This process keeps track of all tokens processed by the element. Received tokens are stored in an ordered list.

Figure 20 shows the dependencies of the Token Pool Manager.

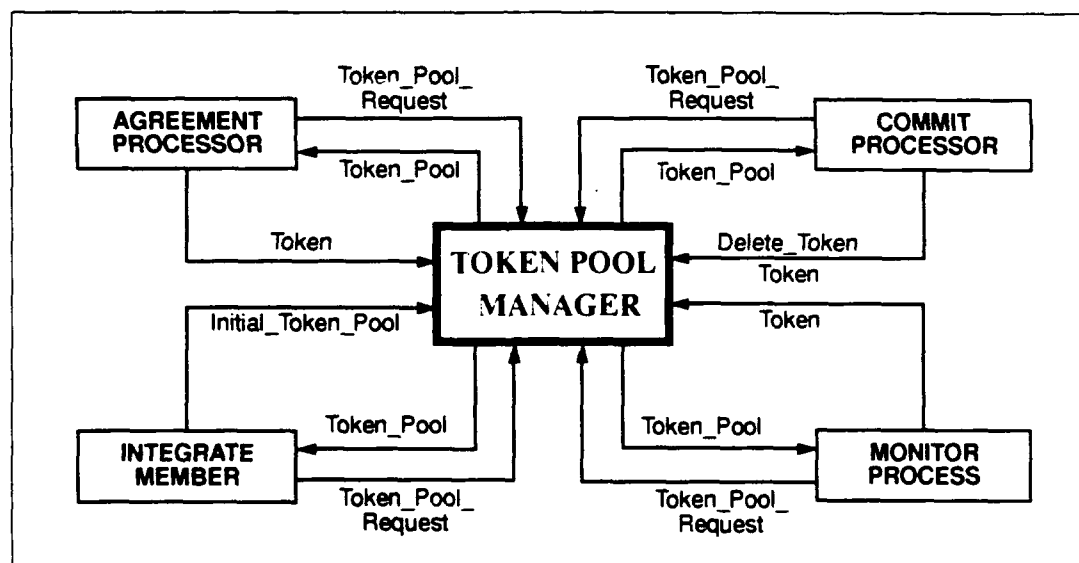


Figure 20 Token Pool Manager- Process Dependencies

When the process initiates its execution, the Token Pool must be initialized as empty. After a successful join an *Initial_Token_Pool* message is received, the Token Pool is extracted from the message.

When a *Token* is received, it is inserted at the end of the list.

The *Delete_Token* message causes the removal of the named token. This is the only mechanism available to delete tokens. Particular care is to be taken, when deleting tokens, to search only for the type and subject, as explained in the discussion of the Agreement Processor.

Input data flows: *Token, Delete_Token, Token_Pool_Request* and *Initial_Token_Pool*.

Output data flows: *Token_Pool*.

C. DATA STRUCTURE DEFINITIONS AND MESSAGE FORMATS

Every element keeps an internal state that reflects the current group as it is perceived. This state is composed of three structures: **Group View**, **Status Table** and **Token Pool**.

In previous sections we described how each structure is managed by a special process, that acts a server. These servers are the Group View Manager, Status Table Manager and Token Pool Manager. The structures are private to the respective server. The Managers provide services to eventual clients (other processes of the member), in order to provide a view of the structure, and the necessary updates.

To carry the state information as needed from process to process, and to carry out the actions required by the protocol definition, processes exchange messages. Some messages are exchanged between members, some others are used exclusively for intra-element communication, and some are used for both situations.

The following sections describe the data structures and the messages used in this protocol implementation.

1. Data structure definition

Each data structure is described in detail.

a. Group View

This structure has three components: **View Number**, **View Size** and **Member List**. A graphical representation of the Group View structure is given in Figure 21.

The view number is an integer value that is incremented by one each time the view changes.

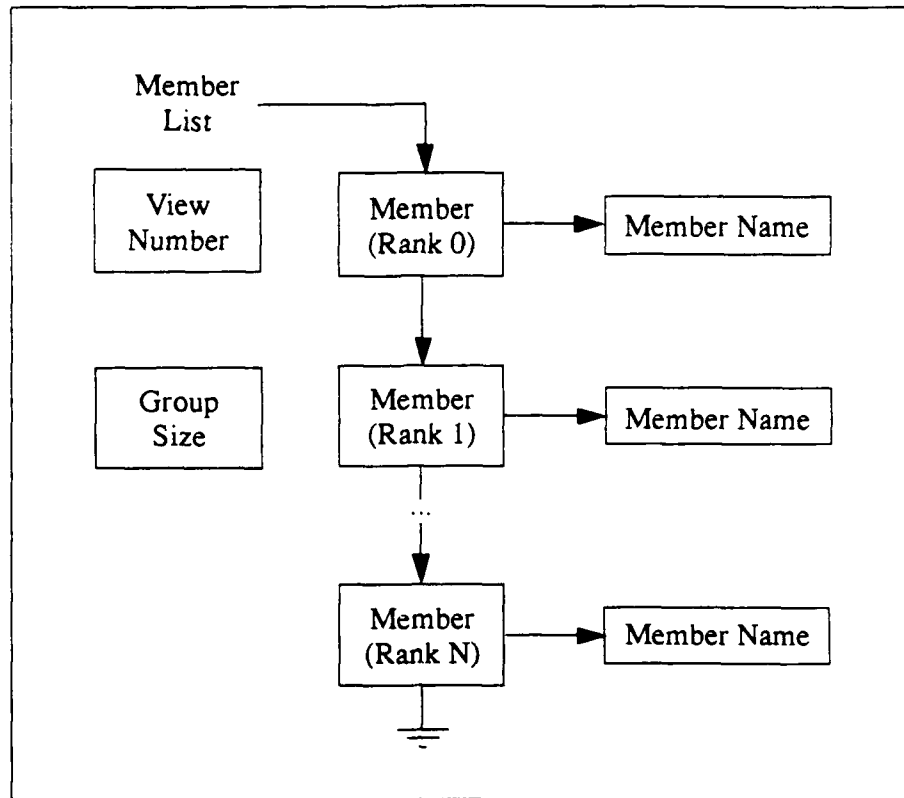


Figure 21 Group View Data Structure

View size is an integer that reflects the number of entries in the member list.

The member list is an ordered data structure that reflects the logical ring that supports the Group Membership Protocol. Each element of the list identifies one particular member. The host is well identified in this structure, because it is always the first entry (i.e., the member with rank 0). Note that the rank information is not stored in this structure, but rather evaluated when needed as described in Chapter II. The structure can be implemented as a linked list, where the host is the node at the head, and all other members are linked by rank order. There is at least one entry in the *Member List* (the current element in a singular group).

Each member has a unique *Member Name* that is used to fully and unequivocally identify it.

b. Status Table

This structure has two components: **Table Size** and **Status List**. The internal structure of the Status Table is given in Figure 22.

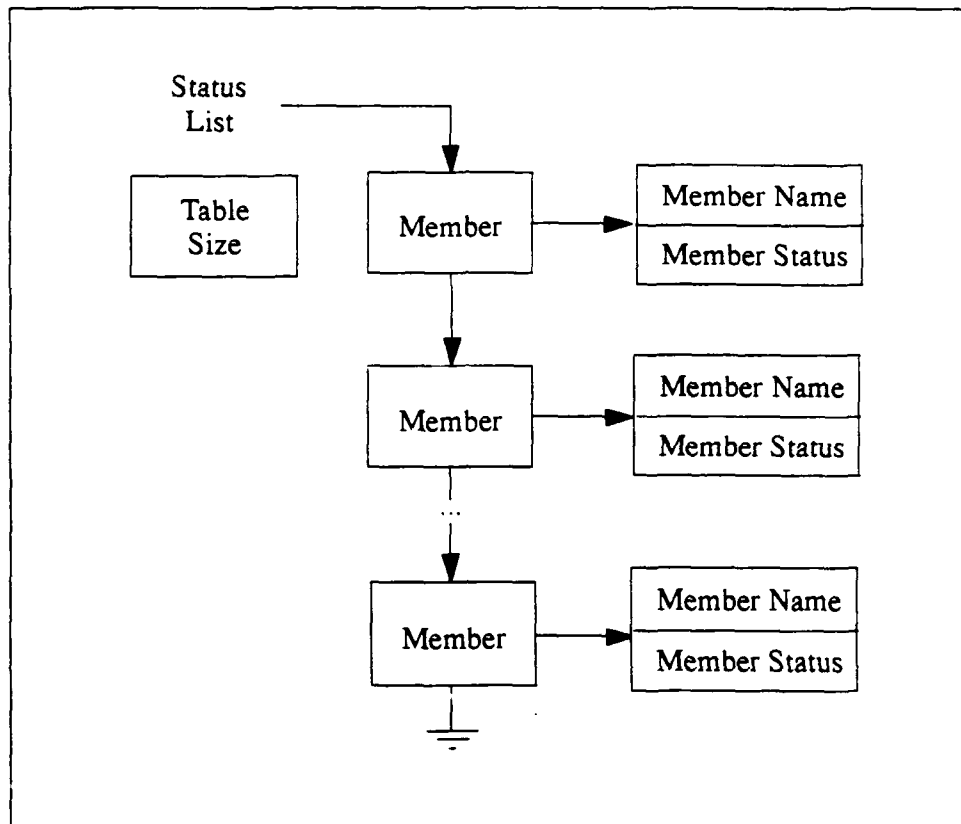


Figure 22 Status Table Structure

Entries in the *Status List* are referenced by the member name, and have a coded string field that reflects the status of the corresponding member.

The *Table Size* field has a value equal to the number of entries in the *Status List*. There can be an empty Status Table, where *Table Size* is zero and *Status List* is a Null list.

c. Token Pool

This structure has two components: **Pool Size** and **Token List**. The structure of the Token Pool is graphically represented in Figure 23.

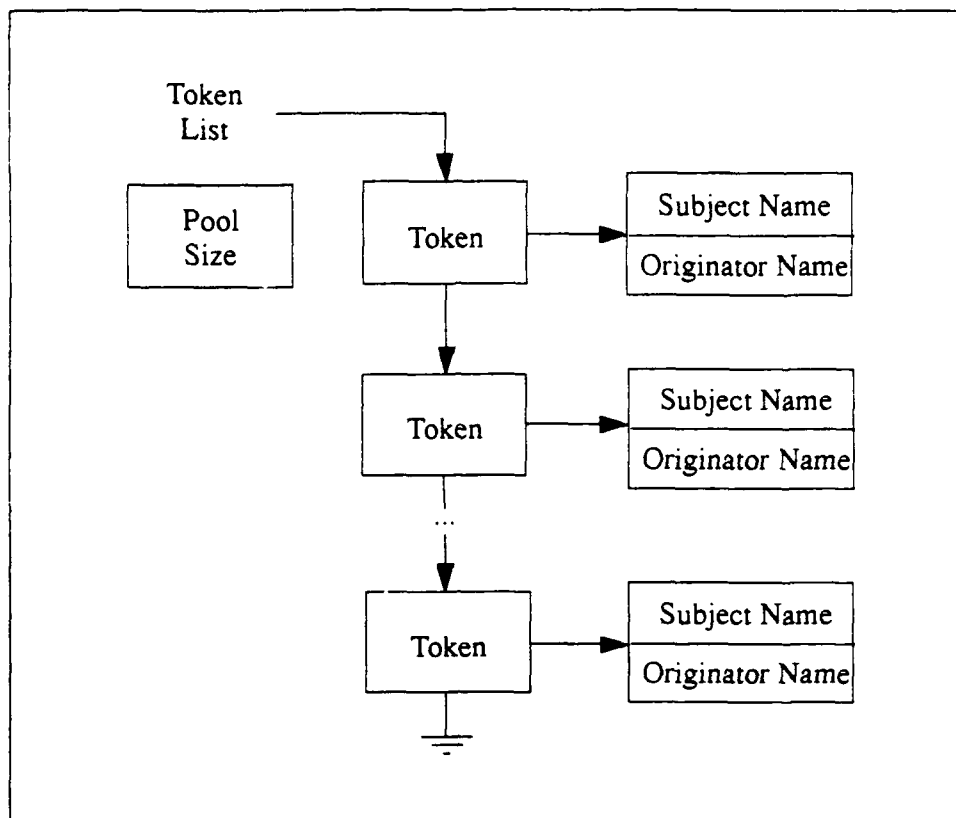


Figure 23 Token Pool Structure

The *Token List* has entries that correspond to tokens. The encoding of a token follows the format $token_type_p(p_k)$, where p_i and p_k are member names of the token originator and subject respectively, stored using the *Element Name* data format.

Pool Size has the value corresponding to the number of entries in the *Pool List*.

The Token Pool can be empty, i.e., there are no unprocessed tokens outstanding. In this case the *Pool Size* is zero and *Token List* is a Null list.

2. Message formats

We have seen that processes can exchange several different messages. These messages were referenced informally during the discussion of the protocol and process specifications, and were identified by descriptive names.

Messages exchanged by the processes that constitute a single element are assumed to use reliable communication channels, i.e., channels that ensure the delivery of messages and maintain

the correct ordering. These messages share a common general format and are collectively named **Internal Messages**.

Figure 24 shows a graphical representation of the internal message format.

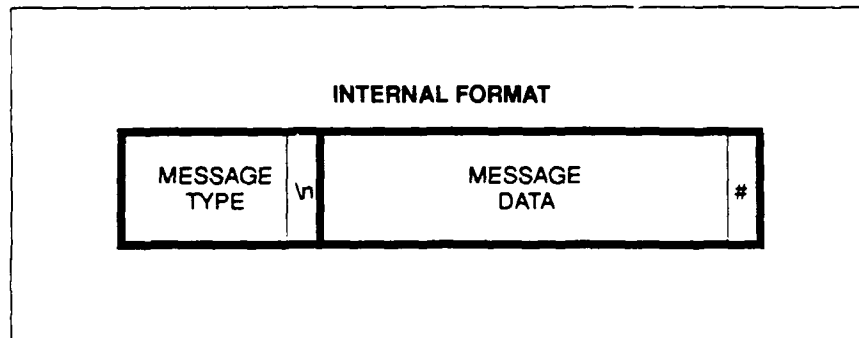


Figure 24 Internal Message Format

Messages that are exchanged between Back and Front processes use the Internet network communication channels. These messages, depending on the adopted network interface, travel through possibly unreliable and non-fifo channels [20]. It is then necessary to encapsulate the message with additional components that are used by the Fifo Channel Layer as described earlier. These messages share a common format and are named **External Messages**.

Figure 25 shows a graphical representation of the external message format.

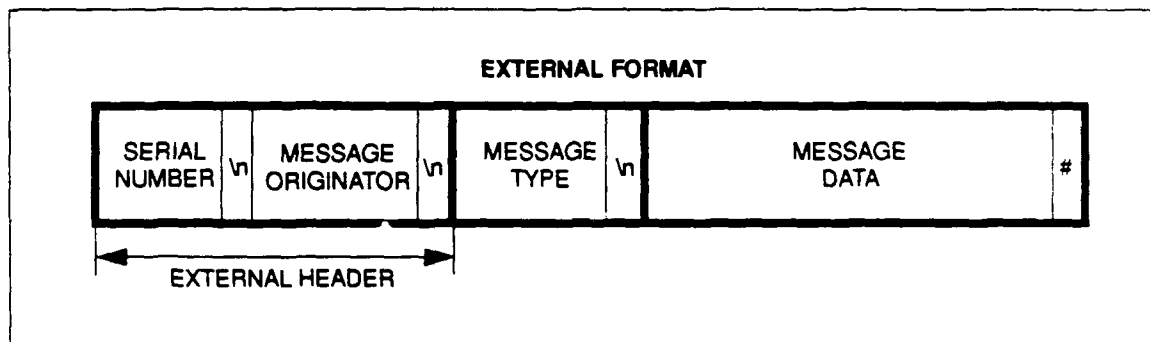


Figure 25 External Message Format

Some messages have only one of the forms, while others have to be encoded in both formats. All fields are encoded as ASCII character strings (non-null terminated) to allow transmission over the standard network interfaces.

Table 3 lists all messages, the coding used in the *message type* field and possible formats. This field has a fixed size of nine characters.

Table 3 MESSAGE LIST

MESSAGE TYPE	MESSAGE TYPE FIELD VALUE	FORMAT	
		INTERNAL	EXTERNAL
Token	tokenokn	X	X
Token Pool	tokenpool	X	X
Initial Token Pool	inittpool	X	
Token Ack	tokenackn		X
Delete Token	delttoken	X	
Initiate Token	inittoken	X	
Status Table	statustbl	X	
Initial Status Table	inittable	X	
Status Query	statusqry	X	X
Status Report	statusrpt	X	X
Group View	groupview	X	
Initial Group View	initgview	X	
Initial Parameters	initparam	X	X
Join Request	joinreqst	X	X
Update Status	updstatus	X	
Update View	updatview	X	
Send Initial Parameters	sndinipar	X	
View Request	viewreqst	X	
Status Table Request	statreqst	X	
Token Pool Request	tokpreqst	X	
Start Timer	starttimer	X	
Timeout	timeout__	X	

As discussed earlier, each member of the group is identified by a unique name, which is used in all messages and internal structures. Each name contains all the information needed to access the member as far as the protocol is concerned, and is referred throughout as Element Name.

Element Name includes the Internet address (IP address) of the host where the member is running, and the IP ports of its Front and Back processes. The IP address is encoded using the standard dot notation ASCII string format [20]. The IP ports are stored as ASCII encoded integers. Since the fields have variable length, a special character ':' is used as a separator.

Figure 26 shows a graphical representation of the Element Name structure.

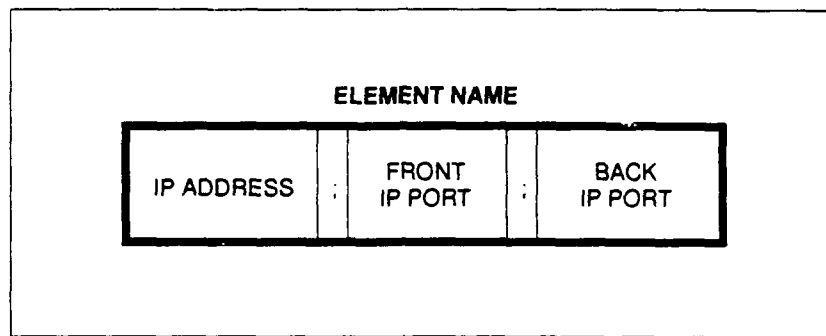


Figure 26 Element Name Structure

Each message type is now described in detail.

a. Token message

This message can be encoded in both internal and external formats. The encoding follows the notation introduced for tokens $token_type_{p_i}(p_k)$, where p_i and p_k are member names of the token originator and subject respectively.

The graphical representation of this message in both formats is given in Figure 27.

The *data* field of this message is just a string representation of an entry on the Token Pool. The *message type* field has the value "tokentoken".

Tokens can have different types, which are listed in Table 4. The *token type* field has the corresponding code, in order to identify the token type. Like the *message type* field, the *token type* has a fixed length of nine characters.

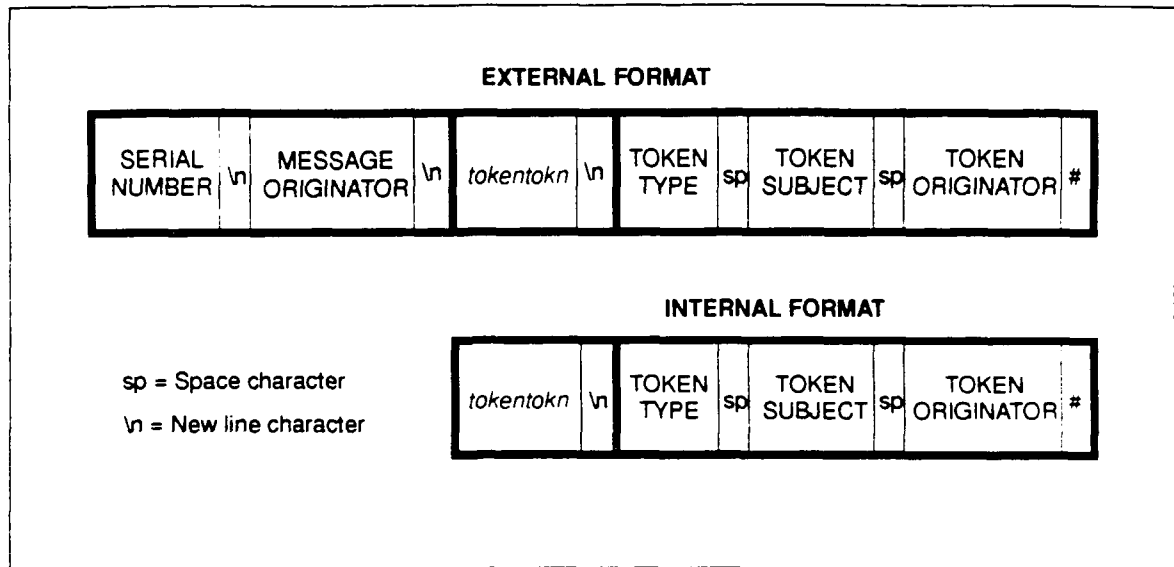


Figure 27 Token Message Format

Table 4 TOKEN TYPES

TOKEN TYPE	FIELD VALUE
Failure agree	<i>failagree</i>
Failure commit	<i>failcomit</i>
Join agree	<i>joinagree</i>
Join commit	<i>joincomit</i>
Join request	<i>joinreqst</i>

b. Token Pool message

This message, like the *Token* message, can be encoded in both internal and external formats. Figure 28 shows a graphical representation of the Token Pool message external format.

The internal format version of this message is similar to the internal version, but excludes the external header, as seen in Figure 29.

The data field is a linear representation of the entire Token Pool structure, starting with the Pool Size. The entries on the Token List are then concatenated with a special character '=' used as a separator.

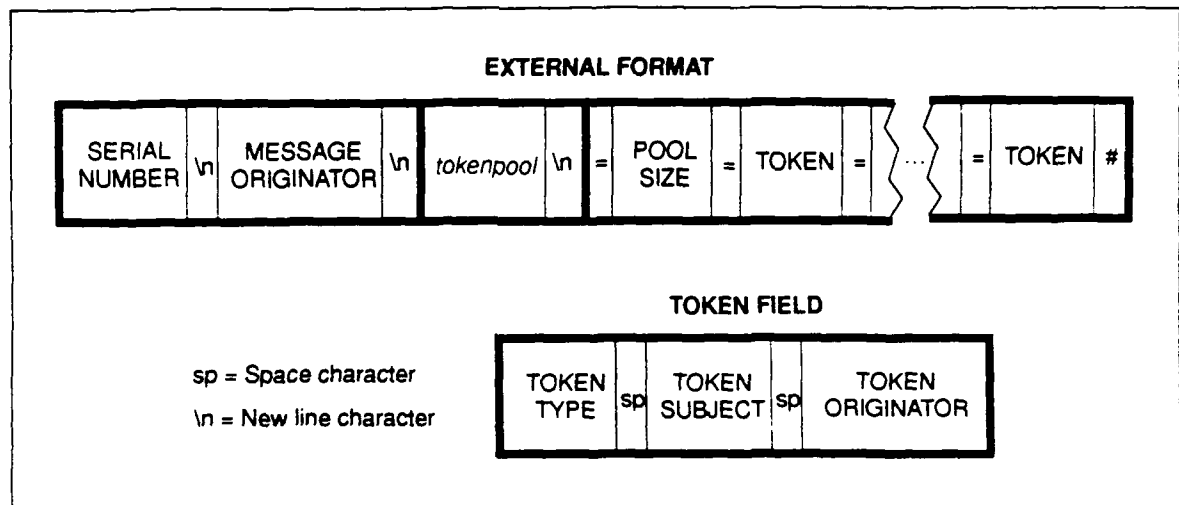


Figure 28 Token Pool Message External Format

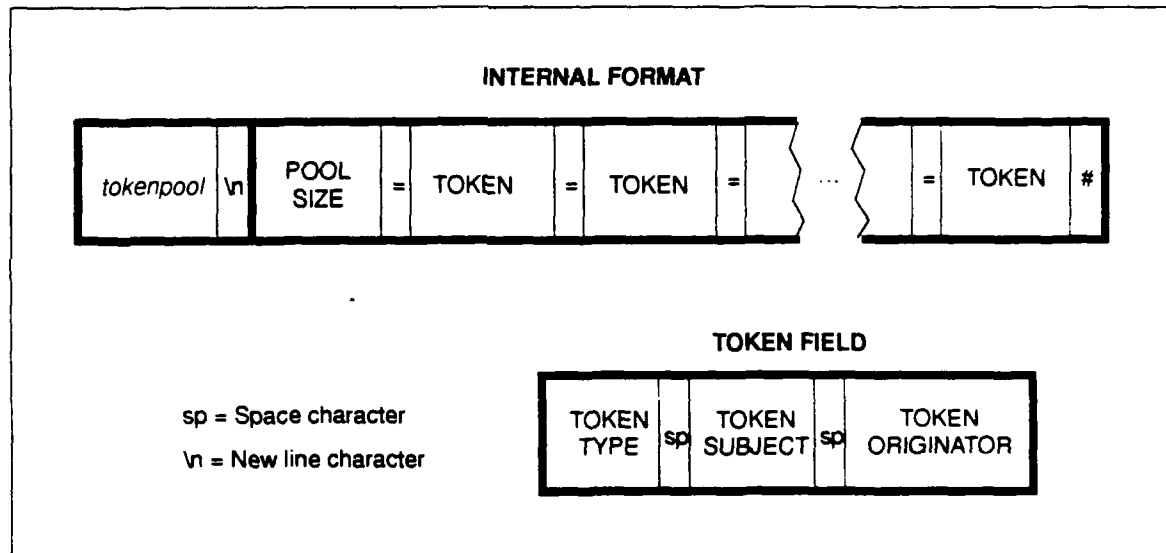


Figure 29 Token Pool Message Internal Format

c. Initial Token Pool

This is an internal message used to initiate the Token Pool when a element becomes a member of a group. Figure 30 shows the structure of this message.

Its structure is in all aspects similar to the *Token Pool* message (internal format), with the exception of the *Message Type* field.

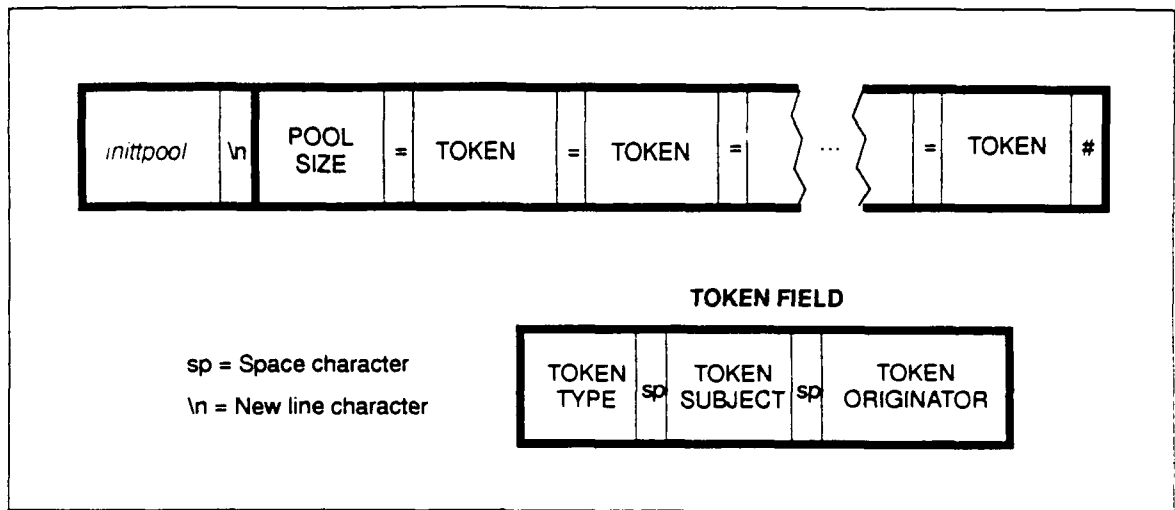


Figure 30 Initial Token Pool Message Format

d. Token Ack message

This message exists only in external format. It is used exclusively to implement the acknowledge protocol at the Fifo Channel Layer level, and is not an integral part of the membership protocol.

Figure 31 shows the graphical representation of this message.

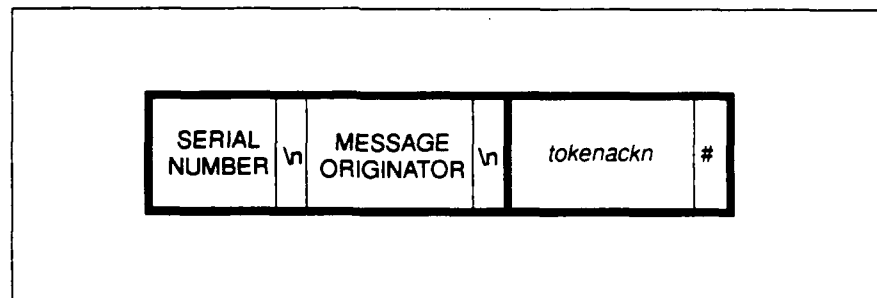


Figure 31 Token Ack Message Format

The serial number is a copy of the serial number of the original message being acknowledged. The message originator field refers to the member that originated the ack message.

There is no data field in this message.

e. Delete Token message

This message only exists in internal format. Figure 32 shows the message format.

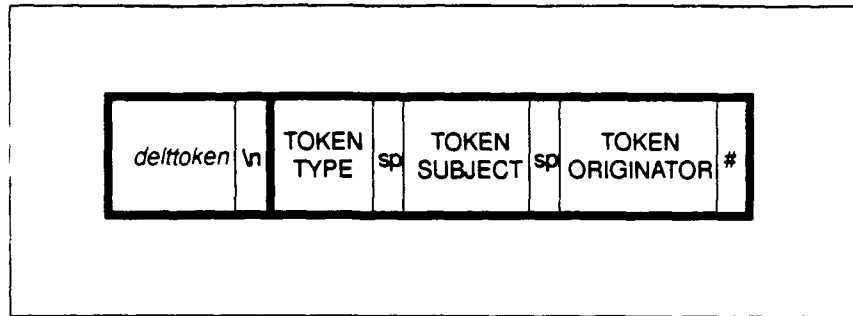


Figure 32 Delete Token Message Format

The data field is equal to the data field of the token message whose token is to be deleted from the *Token Pool*.

f. Initiate Token message

Figure 33 shows the format of this message.

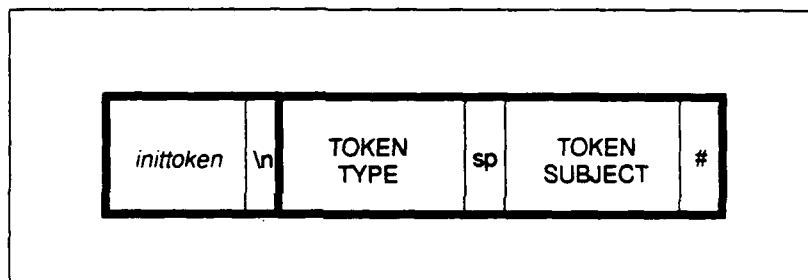


Figure 33 Initiate Token Message Format

This messages is used to signal the Commit Processor and Agreement Processor, to generate and process the corresponding token (commit or join_request/agree, respectively), whose subject is the element named in the token subject field, and with the type described in the token type field.

g. Status Table and Initial Status Table messages

The format of these two messages is depicted on Figure 34. These messages are used to represent the Status Table structure. The Status Table message is assembled by the Status Table Manager and represents the current structure. The Initial Status Table is received by the same manager and is converted into the new Status Table.

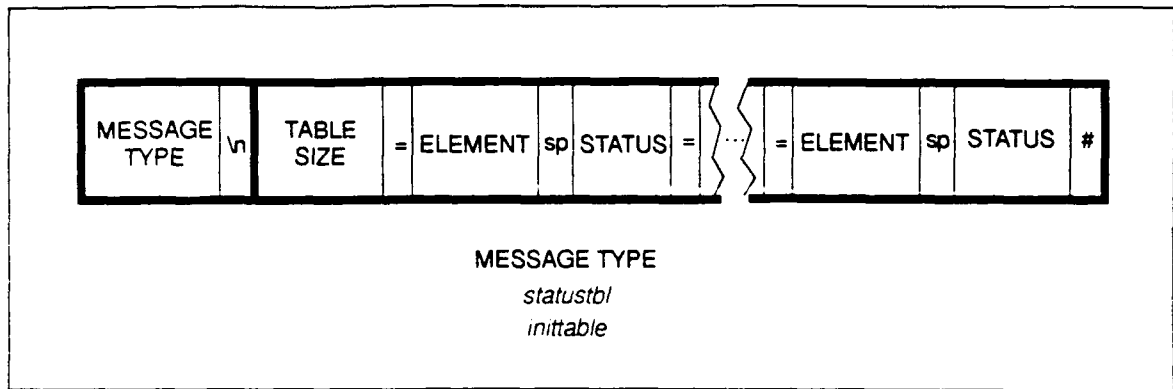


Figure 34 Status Table and Initial Status Table Message Format

The possible status of an element, and the corresponding Status field coding, are listed on Table 5.

Table 5 STATUS TYPES

STATUS	FIELD VALUE
Join requested	<i>joinrqstd</i>
Joining pending	<i>joinpendg</i>
Join agree	<i>joinagree</i>
Fail pending	<i>failpendg</i>
Fail agree	<i>failagree</i>
Clear status	<i>clrstatus</i>

These messages have exclusively the internal format. The data field is a linear representation of the entire Status Table structure, starting with the Table Size. The entries on the Status List are concatenated in order, separated by the special character '='.

h. Group View and Initial Group View messages

These messages are similar in their structure, and are represented in Figure 35

Both messages represent the Group View structure. The Group View message is assembled by the Group View Manager to represent the current view. The Initial Group View is

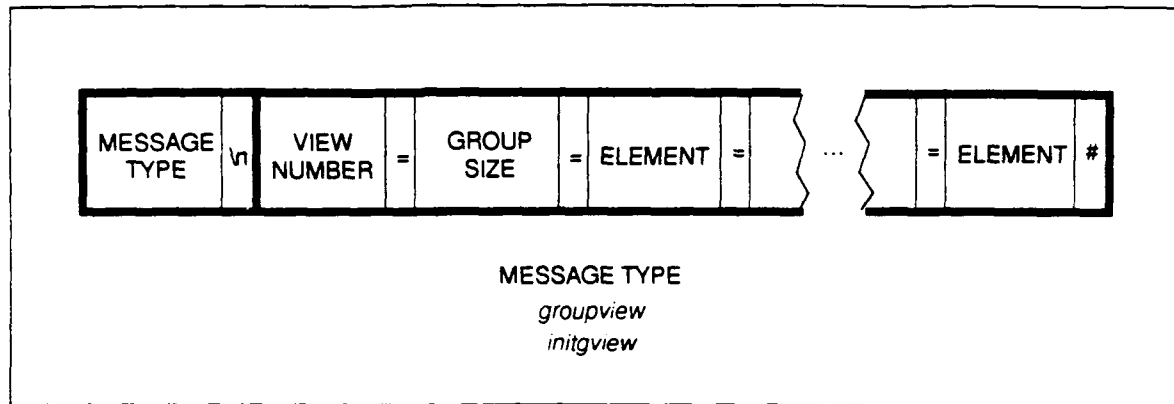


Figure 35 Group View and Initial Group View Message Format

received by the same manager, at the end of the joining procedure, and is used to establish a new Group View.

These messages are exchanged only between processes of the same element, and so they have an internal representation only.

The data field is a linear representation of the Group View structure. The first block duplicates the View Number, and is followed by the Group Size. Then the entries of the View List are concatenated in order. All these blocks are separated by a '=' character.

Elements are represented as usual in Element Name format.

i. Initial Parameters message

This is the more complex message used in this implementation. It can take the internal or external format. Figure 36 shows the internal format only. As specified before, the external format has the same data field, adding only an external header.

The *Initial Parameters* message is sent by the host to the newly joined member. The new member extracts from this message all the information needed to initialize its internal data structures (Status Table, Group View and Token Pool).

As seen in Figure 36, this message is composed of the ensemble of the data fields of the *Group View*, *Status Table* and *Token Pool* messages. These sub-fields are separated by a '@' character.

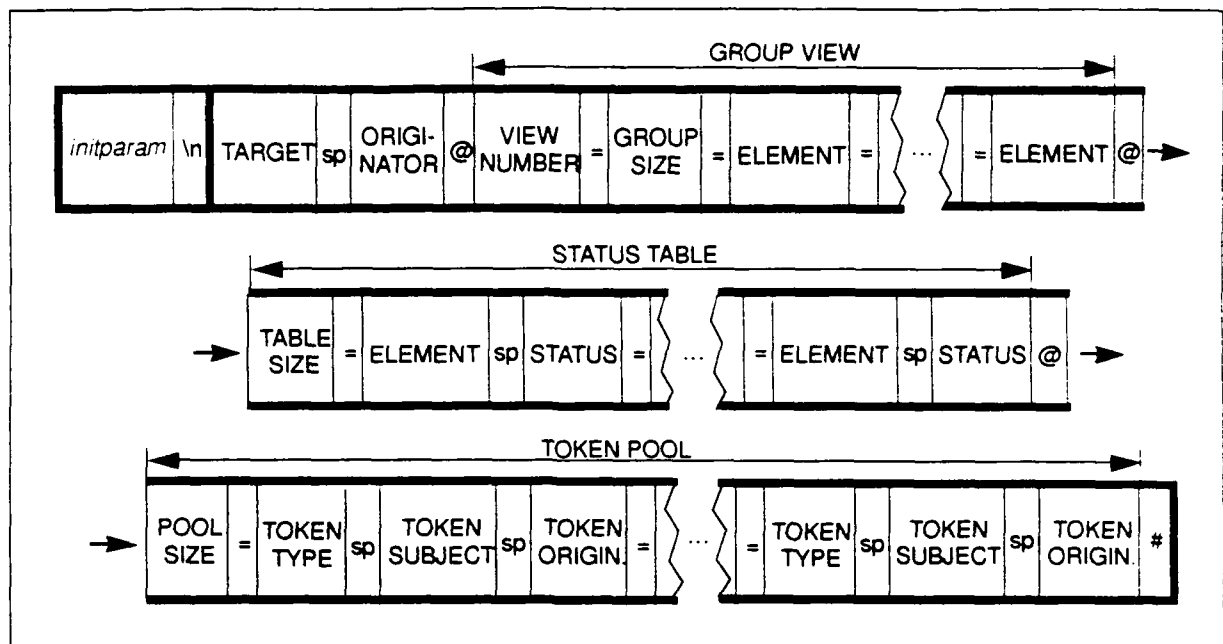


Figure 36 Initial Parameters Message Internal Format

The process who assembles this message is the Join Process of the group host. It uses the current Group View, Status Table and Token Pool.

Integrate Member is the process who disassembles the message and sends the sub-fields to the data managers.

This message is used by the Join Process of the prospective new member to determine that the joining procedure was successfully completed.

j. Join Request, Status Query and Status Report messages

Figure 37 shows a representation of these messages. Note that the only difference is on the message type field.

These three messages have both external and internal forms.

In these messages, *Target* refers to the member to whom the message is addressed to, and *Originator* refers to the element that initiated the message (i.e., the element who sends the message). Both fields are encoded using the *Element Name* format.

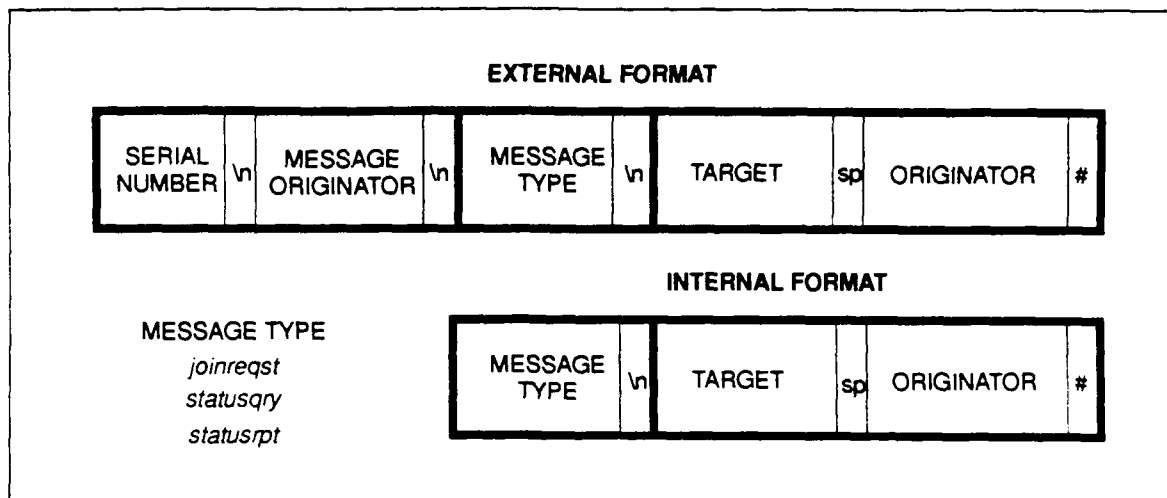


Figure 37 Join Request, Status Query and Status Report Message Formats

k. Update Status and Update View messages

The representation of these two messages is depicted in Figure 38.

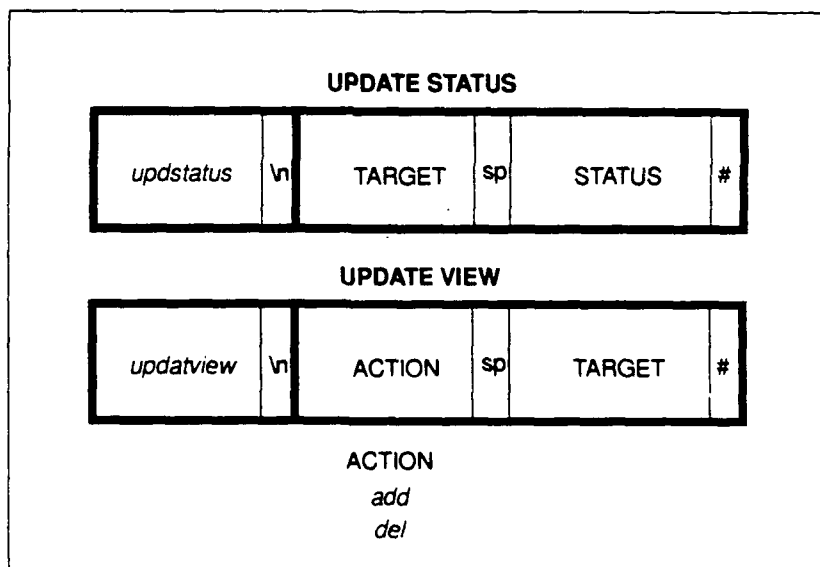


Figure 38 Update Status and Update View Message Format

These messages are used to signal a change in the corresponding internal data structure.

In the case of the *Update Status* message, the Status field could have any of the values listed in Table 5.

The possible Action values of the *Update View* message are listed in Figure 38.

l. *Send Initial Parameters message*

This message is sent by the Commit Processor to the Integrate Member process, requesting it to assemble the *Initial Parameters* message to the specified target (which is the joining element).

Figure 39 shows a graphical representation of this message.

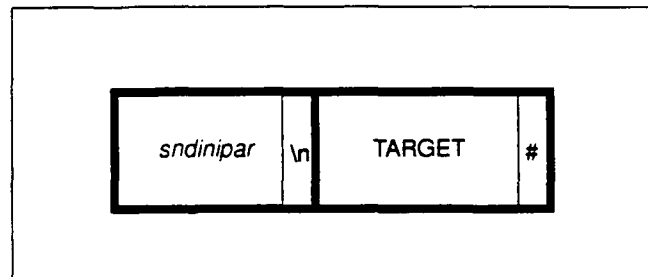


Figure 39 Send Initial Parameters Message Format

m. *View Request, Status Table Request and Token Pool Request messages*

All these messages are similar in format, in the sense that they all have an empty data field. They are 'action' messages, i.e., the information carried by the message is contained in its own type.

A collective representation is shown in Figure 40.

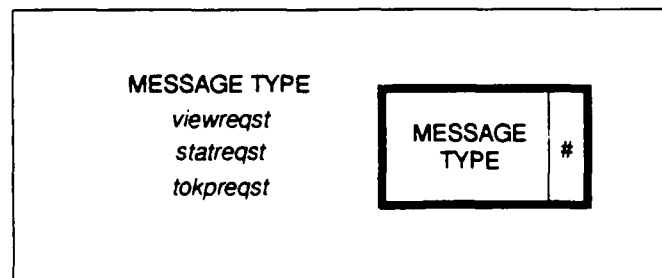


Figure 40 View Request, Status Table Request and Token Pool Request Message Format

n. *Start Timer and Timeout messages*

These messages are exchanged between the Status Monitor and the Timer sub-processes, and between the Join Processor and Join Timer processes.

Figure 41 shows a graphical representation of the messages.

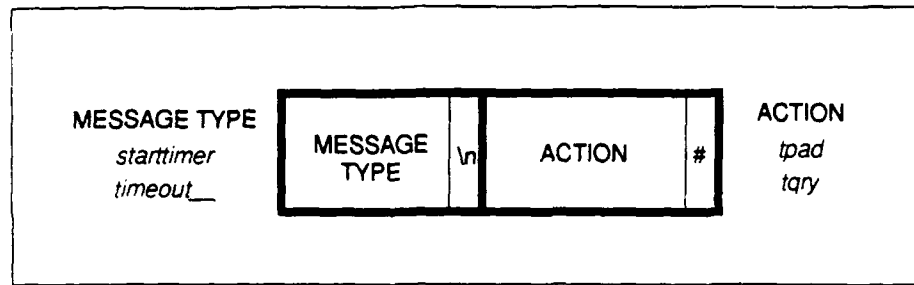


Figure 41 Start Timer and Timeout Message Format

IV. IMPLEMENTATION ON UNIX-BASED MACHINES

In this chapter, we discuss the issues related to the implementation of the protocol on UNIX-based machines. The current implementation was developed over a local area network (Ethernet) of SUN workstations, running SunOS 4.1, which is a Unix flavor that incorporates features from both BSD and System V network and inter-process communication facilities.

A. COMMUNICATIONS LAYER

1. Network Access Protocol

Communication links to/from the application are exclusively intra-host, while different members can reside in one or more hosts. Different members can belong to hosts in a single LAN or they can spread over a WAN.

Communication within an element and between elements of a group has to be reliable and delivered in a strict first-in-first-out order, as required by the protocol.

All these characteristics suggest some desired features for the MP design, namely: a consistent Application Program Interface (API), usable across different network configurations, and adaptable to the distinct classes of links; a minimum of communication links, in order to preserve resources; availability of the group member's address must be available to newly formed elements.

To satisfy these constraints, we used the 4.3BSD implementation of sockets to implement all communication links. The Berkeley socket interface is widely available and constitutes a *de facto* standard, making this implementation highly portable. Links between elements are established using the Internet Domain Protocol to allow transparent communication across networks. Links between processes of an element are implemented with Unix Domain Protocol, which is a simplified version of the Internet Domain Protocol. Apart from the link formation process, the access process is similar for both protocols, making the coding process relatively uniform.

The other possible choice is System V TLI interface. This is a more recent standard, that follows the OSI layered architecture model. Being a newer standard, it is currently not as popular as BSD sockets, but is expected to become dominant in the future. Since these two interfaces have a similar programming interface structure, as seen in Table 6, it would be relatively easy to port an

Table 6 COMPARISON OF SOCKETS AND TLI (ADAPTED FROM [20])

Process Personality	Action	Socket	TLI
Server	allocate space		t_alloc()
	create endpoint	socket()	t_open()
	bind address	bind()	t_bind()
	specify queue	listen()	
	wait for connection	accept()	t_listen()
	get new fd		t_open() t_bind() t_accept()
Client	allocate space		t_alloc()
	create endpoint	socket()	t_open()
	bind address	bind()	t_bind()
	connect to server	connect()	t_connect()
	transfer data	read() write() recv() send()	read() write() t_rcv() t_snd()
	datagrams	recvfrom() sendto()	t_rcvudata() t_sndudata()
Common	terminate	close() shutdown()	t_close() t_sndrel() t_snddis()

application written in one of them to support the other. The current implementation of the protocol

is designed using a structured approach, in which access to the network interface is hidden in the lowest level software layer that concentrates the majority of the standard-conversion related issues.

a. Inter-member communications

Having settled for the BSD socket interface for all communication links, and given that the protocol must use Internet based hosts, thus utilizing the TCP/IP protocols to access the network, a decision had to be made concerning the type of transport layer interface to use.

TCP/IP provides two transport layer protocols: TCP and UDP.

TCP, also referred to as TCP/IP, implements a connection-oriented, reliable, full-duplex, byte-stream service for message delivery [20].

UDP, also known as UDP/IP, on the other hand, provides a connectionless, unreliable datagram service [20].

Table 7 summarizes the characteristics of both protocols.

Table 7 COMPARISON OF FEATURES FOR UDP AND TCP (FROM [20])

	UDP	TCP
connection-oriented	no	yes
message boundaries	yes	no
data checksum	opt.	yes
positive ack.	no	yes
timeout and rexmit	no	yes
duplicate detection	no	yes
sequencing	no	yes
flow control	no	yes

Apparently, TCP would be the first choice for our inter-member communications (those that have to travel across the network), since it provides reliability, and also ensures FIFO message delivery. Unfortunately, if such a connection is established, and one of the end-points

hangs in the middle of the communication process, it is possible that the other end-point would hang also, without possibility of recovery using the regular access mechanisms.

UDP datagrams are inherently unreliable (that is, they might not be delivered at all, delivered after an arbitrary delay, or even delivered in multiple copies). This means that it is necessary to add an extra mechanism to the FIFO Channel Layer, that provides the reliability and fifo ordering to messages delivered. This mechanism was implemented as described in Chapter III. The main benefit of the additional overhead is that UDP totally decouples the communicating end-points, making the message exchange asynchronous, and isolating failures such that failure of one end-point would not bring the other one down. This is a crucial property for the MP implementation, since member failures constitute central events, and have to be confined to the affected member only.

b. Intra-member communications

Sockets provide an Unix-domain protocol, that is restricted to inter-process communications within a single Unix system (typically at the host level). This protocol has an access interface in all respects similar to the Internet-domain version. It also provides a choice between connection-oriented and connectionless interface. Unlike the Internet domain version, both these interfaces are considered reliable.

The connection-oriented Unix protocol provides flow control, while the connectionless protocol provides datagram delivery without any flow control. Since it is possible for a datagram client to send data so fast that buffer starvation can occur, it is recommended that only connection-oriented Unix domain protocol be used [20].

The problem of an end-point tying a connection in the case of a failure, is not relevant, because such a failure has to bring the entire element down anyway.

2. Socket Access Abstraction

To encapsulate all socket access issues, and thus delivering a higher level interface to the message handling process, a toolbox was developed and access to socket facilities restricted to entries in it. This toolbox is called 'SOCUTIL.C' and is listed in the appendix. Message handling

functions were also developed, and are intended to be used in close connection with these socket access routines. These message handling functions are grouped on a toolbox named 'MSGUTIL.C'.

a. Unix-domain socket access

All processes in this MP use a standard sequence of calls to establish a working unix-domain socket where incoming messages are received and responses are transmitted back, when applicable. This sequence is exemplified in Figure 42

```
1 strcpy(socpath, UNIXSTR_TMPL);
2 mktemp(socpath);
3 socfd = createUN(socpath);
4 listen(socfd,5);
5 clen = sizeof(caller);
6 newsocfd = accept(socfd, (struct sockaddr*)&caller, &clen);
7 revmsglen = readmsg(newsocfd, &msg, eom);
8 ...
9 txmsglen = writemsg(newsocfd, msg, msglen);
10 close(newsocfd);
11 ...
12 close(socfd);
13 unlink(socpath);
```

Figure 42 Sequence of Calls to Establish a Working Unix-domain Socket

Lines 1 and 2 generate an unique pathname that will be used to create the socket. Unix-domain sockets are implemented in the OS kernel as file system entries, and so they have 'real' file names (actually they appear in the file system as a zero-size file). UNIXSTR_TMPL is a filename template of the form '/tmp/so.XXXXXXX', where the Xs represent wildcards that are filled in by the operating system. Note that the size of this string has to be fixed to 14 characters due to an OS glitch [20].

Line 3 calls a function on 'SOCUTIL.C' that creates the socket and returns a file descriptor that is used as an handler, in the same way as for a regular file.

Line 4 establishes a queue that stores incoming messages in a fifo order. This allow for the process to receive messages in an asynchronous way.

Line 5 blocks the execution waiting for an incoming message to arrive at the socket. When this happens, a copy of the socket, where that one message is ready to be read is returned. The original socket is made available to accept more messages, without interfering with the processing of the current message.

Lines 6 and 8 perform the reading and writing of messages, respectively, to this new socket, by calling a pair of functions from 'MSGUTIL.C'.

The block of code that includes lines 5-9 is typically run in a closed loop, waiting for new messages, and servicing them. This makes the process that runs this code a server, in the socket message exchange process.

Lines 11 and 12 illustrate the way a socket attached to a well known location is to be discarded. The file descriptor has to be closed, and the file system entry has to be removed (unlinked).

The sequence of calls used to establish a connection with a socket of another process is illustrated in Figure 43.

```
1 newsocfd = connectUN(rmtsocpath);  
2 txmsglen = writemsg(newsocfd, txmsg, txmsglen);  
3 ...  
4 rcvmsglen = readmsg(newsocfd, &rcvmsg, eom);  
5 close(newsocfd);
```

Figure 43 Sequence of Calls to Connect to a Remote Unix Socket

Line 1 calls a function of 'SOCUTIL.C' to create a temporary socket where the message exchange takes place. The filename of the remote socket has to be known to the calling process.

Lines 2 and 4 perform the message writing and reading to/from the socket.

Once the message is completely processed, the new socket is no longer needed, and should be discarded (line 5).

b. Internet-domain socket access

Although none of the processes of the MP uses exclusively Internet-domain sockets to communicate with other processes, the access sequences for this situation are described. Part of these sequences will be used by the 'FRONT' and 'BACK' subprocesses when operating the Internet ports.

To establish a socket where messages are received and, if applicable, where responses are sent to, the sequence of calls listed in Figure 44 is used.

```
1 IPport = htons(0);
2 socfd = createUDP(&IPport);
3 rcvmsglen = recmsg(socfd, &rcvmsg);
4 ...
5 txmsglen = senmsg(txmsg, txmsglen, IPaddr, IPrtport);
6 ...
7 close(socfd);
```

Figure 44 Sequence of Calls to Establish a Working Internet Socket

Line 1 establishes the IP port to be used by the socket. This could be a non-zero value, in which case the given number would be used. If the port number is set to zero, the operating system will provide one available port.

Line 2 creates the socket and attaches it to the specified IP port. The actual IP port where the socket is attached is returned. This is used to capture the port when it is assigned by the OS (i.e., when a zero port was specified).

Lines 3 and 5 perform the message writing and reading, using functions from 'MSGUTIL.C'. This block is typically executed in a closed loop, receiving and servicing incoming messages. Note that *recmsg()* is a blocking call (i.e., it blocks the process execution until a message arrives at the port).

To connect to a remote port, and send a single message, the sequence of actions depicted in Figure 45 is to be taken.

Note that since the Internet port uses datagrams (it works with UDP), and considering that it is not known if the receiver is operative, it does not make sense to read a message in response

```
1 port = htons( (u_short)rmtport );  
2 senmsg(msg, msglen, IPaddr, port);
```

Figure 45 Sequence of Calls to Connect to a Remote Internet Socket

using the same port. To receive a response, this should arrive at the local well-known port (Figure 44).

c. Multi-port socket access

It is possible to mix both Unix-domain and Internet-domain sockets in the same process. This allows us to create multi-port processes that receive messages in any of the open sockets, regardless of their type, and regardless of the order of arrival of messages.

FRONT and BACK sub-processes use this facility to provide the interface for messages that have to cross the boundary of a given element.

Figure 46 lists the sequence of actions to create a multi-port facility.

Lines 1-6 create a socket for each port. Lines 7-9 prepare a flag that signals which ports are to be read (in this case both ports). Line 10 blocks execution until one of the ports receives a message. There are no ordering or sequencing constraints on the arrival of messages at any of the set ports.

Line 11 determines if the socket that has a message is the Unix socket. Lines 12-16 process the incoming message.

Line 17 determines if the socket that has a message is the Internet socket. Lines 18-20 process the incoming message.

The complete listings for FRONT and BACK subprocesses are presented in the appendix. These processes use functions available in another toolbox, 'FIFOUTIL.C'. This toolbox includes functions that implement the token queue used in FRONT to serialize outgoing tokens, as explained earlier, during this subprocess specification.

```

1  /* create Unix port */
2  strcpy(socpath, UNIXSTR_TMPL);
3  mktemp(socpath);
4  unsocfd = createUN(socpath);
5  listen(unsocfd,5);
6  /* create Internet port */
7  IPport = htons(0);
8  IPsocfd = createUDP(&IPport);
9  /* set desired ports to be read */
10 FD_ZERO(&fdread);
11 FD_SET(unsocfd, &fdread);
12 FD_SET(IPsocfd, &fdread);
13 /* wait for any of the ports to receive a message */
14 select(32, &fdread, NULL, NULL, NULL)
15 if(FD_ISSET(unsocfd, &fdread)){
16     /* Unix socket ready */
17     clen = sizeof(caller);
18     newsocfd = accept(unsocfd, (struct sockaddr*)&caller, &clen);
19     rcvmsglen = readmsg(newsocfd, &msg, eom);
20     ...
21     txmsglen = writemsg(newsocfd, msg, msglen);
22     close(newsocfd);}
23 if(FD_ISSET(IPsocfd, &fdread)){
24     /* Internet socket ready */
25     rcvmsglen = recmsg(IPsocfd, &rcvmsg);
26     ...
27     txmsglen = senmsg(txmsg, txmsglen, IPaddr, IPmtport);}
28 ...
29 close(unsocfd);
30 unlink(socpath);
31 close(IPsocfd);

```

Figure 46 Sequence of Actions to Create a Multi-port Facility

B. APPLICATION INTERFACE AND HIERARCHICAL STRUCTURE

Figure 9 shows that there is a connection from the application to the MP instance. This connection represents the action of creating and invoking an instance of the MP on the local host. There is another connection, in the case a communication link, that is used to deliver the current group view from the MP instance to the application.

The hierarchical program execution structure is presented in Figure 47.

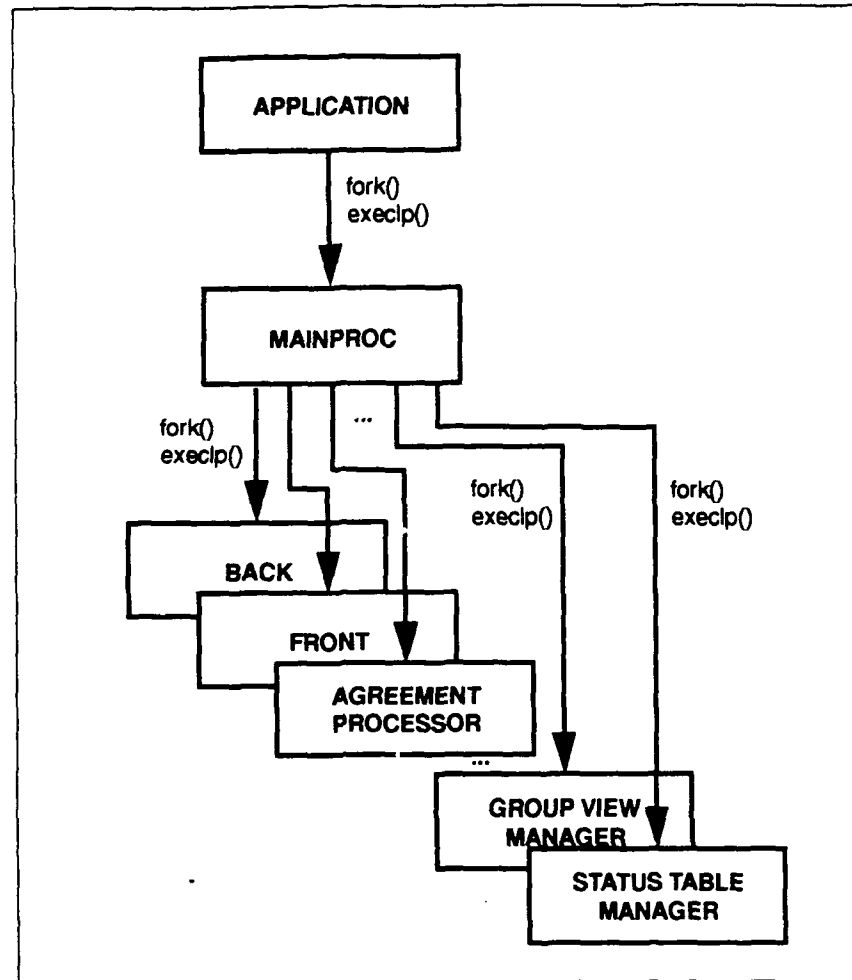


Figure 47 Hierarchical Program Execution Structure

The application creates a socket where it is going to receive the latest group view from the MP instance. Then it executes a *fork()* system call, in order to create a copy of itself (a *child process*), running concurrently. The *child process* executes then an *execvp()* call that transfers control to the program 'MAINPROC', passing it the name of the group that the newly created element is supposed to join (or initiate), the list of sites where to search for existing members, and the address of the socket where it wants to receive the group view. The original program (i.e., the *parent process*) is still executing the application, and it monitors the socket for incoming messages from the MP.

'MAINPROC' is the process responsible for the initialization of the MP instance. It creates all sockets, initializes them and executes a succession of *fork()* and *execvp()* calls, thus starting all processes of the MP. Each new process receives, in its invoking command line argument list, the addresses of all sockets it needs to contact during its life cycle, and all other relevant information (such as the complete element name of the current MP instance, the name of the group, the list of sites to search, etc.), in an as-needed basis.

To test the creation process, and the interaction of the application and the MP instance, an example application program was created, and named 'SIMPLEAPP.C'. The source code listings of 'SIMPLEAPP.C' and 'MAINPROC.C' are presented in the appendix.

A mechanism to allow the application to force the shutdown of the element is provided, with the use of system signals. The same mechanism is used by the MP to inform the application of a failure or departure. A failure represents a crash of one or more processes, or an internal error condition. A departure occurs when the member is still fully operational but was assessed as failed by the group, and thus departed from it. In this situation the member receives the agreement token for its own departure, recognizes it and shuts itself down in a benign way.

V. PERFORMANCE AND EXTENSION ANALYSIS

In asynchronous distributed environments, the number of messages required to complete an action indicates the cost in terms of response time and system resources. In controlled environments (such as a single local area network (LAN)), it is possible to exploit the low-level features such as hardware multicast to reduce the message complexity. Nonetheless, for portability and scalability reasons, such features are not integral part of the basic software implementing the protocol. While such features can always be used for performance tuning, the basic protocols must be analyzed in terms of the number of point-to-point messages.

In this chapter, message complexity for the decentralized approach of the proposed MP is compared with that reported in [5] for a centralized approach to the GMP. For the stated reason, the analysis given below assumes only point-to-point messages (no hardware multicast). First, the complexity for a single departure is given for a group spread over a single LAN (see Figure 48), then for a group spanning several LANs, spreading over a wide area network (WAN) (see Figure 49). The worst-case of a string of departures is then considered. Scalability issues are discussed in close connection with the results obtained.

A. MP OVER A LAN

Let there be n members in the group depicted in Figure 48, before a single departure.

In [5], when this departure is not that of the coordinator, the MP requires one message for detection of the *faulty* status, and $(n-2)$ messages for each of the *exclude*, *response* and *commit* rounds. Thus a total of $(3n-5)$ total messages are required. By using a compressed update, this is reducible to $(2n-3)$. When the coordinator itself departs, this protocol requires one message for *faulty* status detection, and $(n-2)$ messages for each of the rounds for *interrogation*, *response*, *proposal*, *response*, and *commit*.

In contrast, the proposed MP requires $(n-1)$ messages for both the *agreement* and *commit* phases, and an extra message to send the token pool, giving a total of $(2n-1)$ messages. Since there is no centralized manager, all departures have the same cost.

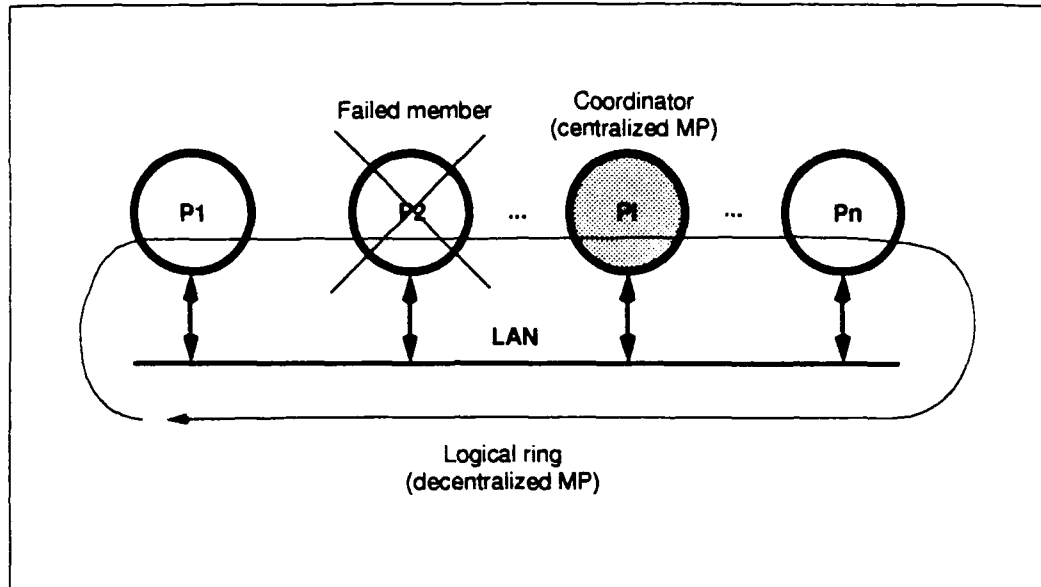


Figure 48 MP Over a Single LAN

For a join, the centralized protocol still requires $(3n-5)$ messages, whereas the decentralized MP may require $(2n+3)$ messages in the best case (one join request message, n for both join agree and join commit, one token pool, and one initial parameters message), and $(3n+2)$ messages in the worst case (same as best-case plus $n-1$ join request tokens). The best case occurs when the joining member locates the ring host, and sends it the join request message. The worst case occurs when the join request message is sent to the clockwise neighbor of the host, and so there is a join request token circulating over the ring until it reaches the host.

Table 8 summarizes these results.

Table 8 MESSAGE COST COMPARISON

	Centralized Protocol		Decentralized Protocol
	Basic update	compressed update	
Departure (non-coordinator)	$(3n-5)$	$(2n-3)$	$(2n-1)$
Departure (Coordinator)	$(5n-9)$	-	
Join	$(3n-5)$	-	$(2n+3)$ to $(3n+2)$

It is to be noted that, in the centralized MP, the coordinator has to maintain communication with all the members throughout the reconfiguration process, thus penalizing one single element of the group, and making the algorithm asymmetric. In the decentralized MP, all members must periodically monitor their anti-clockwise neighbors. This results in a spreading of the constant monitoring overhead, and a quasi-symmetric algorithm (the exception is for the host of the ring during the join process, who has a special responsibility, but not an extended functionality).

When the centralized MP is used, it will be invoked when some higher level process or multicast attempts to communicate with a failed member. When this attempt fails, the coordinator is informed and the MP is invoked for a consistent change to the membership at all operational members. Until that change takes place, the application process is monitored.

In the centralized MP, the constant monitoring makes the latest membership view available as a service. This effectively decouples the membership service from the higher level of software which uses the MP services. This constitutes a significant benefit of this approach.

Considering the costs of the protocol execution, we can conclude that the proposed decentralized protocol has good scalability, since the cost increases linearly with the number of elements, and at a lower rate than the centralized approach.

B. MP OVER A WAN

Consider a group that is spread over multiple LANs connected by a gateway, as pictured in Figure 49.

Typically, messages between members in different LANs will take longer than messages exchanged within a LAN, since the gateway is a shared resource and may have to perform message format transformation and routing functions.

Let k out of a total of n members be in a LAN that also includes the coordinator. For a departure of a member other than the coordinator, or for a join, the centralized MP requires $3(n-k)$ messages across the gateway. When the coordinator departs, $5(n-k)$ messages must cross the gateway.

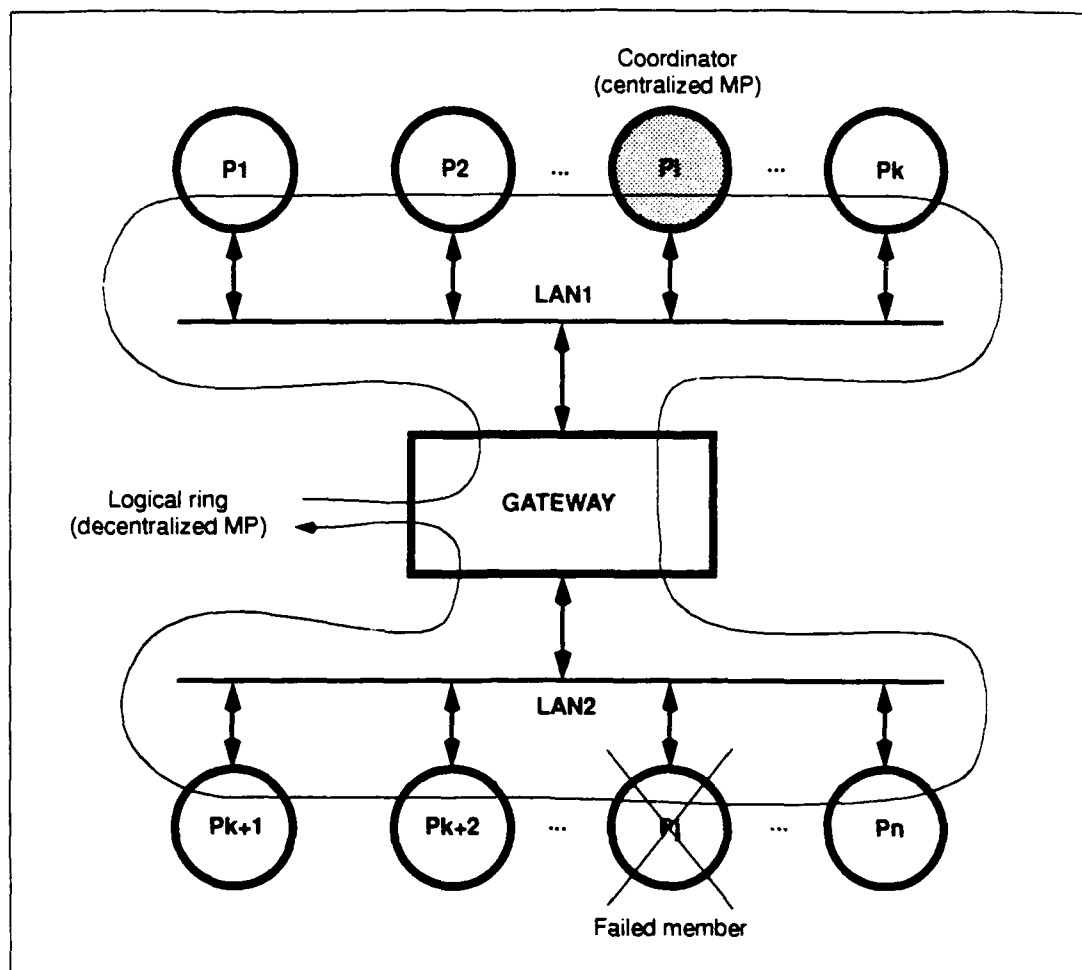


Figure 49 MP Over a WAN

The proposed decentralized MP offers significant savings in this case: only 4 messages must cross the gateway for a departure, and 6 for a worst-case join. The two status-query/response pairs of messages that must cross the gateway for periodic monitoring, can have their period and time-outs tuned to their particular situation.

We can conclude that the ring configuration offers significant benefits for operation over a WAN. These benefits are even more evident when the number of gateways increases.

C. STRING OF MEMBERSHIP CHANGES

When successive reconfiguration attempts are interrupted by a string of departures, the centralized MP has a message complexity of $O(n^2)$. The proposed MP also requires $O(n^2)$ messages as shown below.

The worst case occurs when members fail successively just prior to propagating the commit token for which they had generated the agreement token. For the first departure, there will be $(n-1)$ agreement messages. The commit phase for this departure is carried out by the clockwise neighbor of the agreement initiator (since it failed just prior to propagating the commit token), costing $(n-2)$ commit messages. In addition, a token pool message is required. This process is repeated until the ring reduces to a single member. Thus the total cost is

$$\sum_{i=1}^{n-2} [(n-i) + (n-i-1) + 1] - 1 = n^2 - n - 3$$

It is emphasized that this MP allows any number of departures, provided that there is no network partition, whereas the centralized MP imposes the condition that less than half the operational members fail.

VI. CONCLUSIONS AND RECOMENDATIONS

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

Joins and departures can be interleaved, since they are processed identically. Since there is no centralized responsibility, the overhead for committing a change is constant at $(2n-1)$, where n is the group size. No special facilities such as broadcast messages, ordered access, synchronized actions, and even reliability and FIFO delivery by the network, are required. The protocol implementation delivers the reliable FIFO network abstraction as required by the core protocol. It is emphasized that, in asynchronous environments, the responsiveness of a MP can only be determined by the number of messages required. Strict time-bounds cannot be derived unless the network places such bounds on the individual message delivery. While total message cost is lower, the message overhead of the proposed MP is shown to be superior to [5], which is the only other group membership protocol that uses a fully connected network of FIFO channels that the author is aware of. It also enables complete decoupling of the membership service from the higher level software desirable for scalability of distributed applications [3].

Unlike the centralized approach, this MP does not make any majority-based decision. Therefore, if the network partitions (in violation of the initial assumptions, but corresponding to common real-world situations), it is possible that a group will remain operational in each partition. This is in contrast to the centralized MP which ensures that the group remains operational only in the partition with majority. Relaxing this no-partition network assumption is one of the future lines of work that could benefit the proposed protocol. One possible approach is to implement a method

to reconcile the application state when the partition ceases. This leads us to another desired extension of the protocol, to include merging of two autonomous groups.

Since *client-server* computing has become increasingly popular, it is desirable to study the adaptability of the proposed MP to such an environment, where client members join and depart frequently. Fully incorporating all clients in the group would penalize the overall responsiveness, and would have the undesirable side-effect of making all client-server sessions sensitive to each other (clients would be aware of each other, and a client failure would invoke a reconfiguration process involving all active clients). An alternative, and apparently better approach would be to create a *core server group*, and as many *virtual groups* as clients, where each virtual group would include all members of the core server group and a single client. Members of the server group would be aware of all virtual groups and would incorporate the methods to correctly reconfigure the groups in a transparent but still consistent way.

For more general applicability of this protocol, such as over different possible network configuration and combinations, a hierarchical distribution of members might prove to be a more effective way of configuring the basic ring. In this kind of configuration, each cluster on a particular hierarchical level would run an autonomous MP, that would interface with all others at the same level. This is particularly well suited to large WANs, where different sections have an homogeneous internal composition, but are dissimilar from each other, and/or rely on slow or costly interconnections.

Work can also be done to incorporate mechanisms capable of using the facilities provided by synchronous environments, mostly present in real-time applications, where efficiency is a major concern.

The current implementation is based on the socket interface provided by the Berkeley Unix implementation (4.3BSD). The programs were developed using modular methodology, and a machine independent approach, thus making it highly portable within Unix flavors that support the socket interface. It would be desirable however, for portability issues, to develop a multi-standard version that would support ATT's Unix System V Release 3 TLI interface, as well as other current and possibly future available standards.

LIST OF REFERENCES

- [1] Flaviu Cristian "Agreeing on who is present and who is absent in a synchronous distributed system," in *Proceedings of the 18th International Conference on Fault Tolerant Computing, Tokyo, Japan*, pages 206-211, 1988.
- [2] Flaviu Cristian, "Understanding Fault-Tolerant Distributed Systems," in *Communications of the ACM*, pages 57-78, February 1991.
- [3] Kenneth Birman, Andre Schiper and Pat Stephenson, "Lightweight causal and atomic group multicast," in *ACM Transactions on Computer Systems*, pages 272-314, August 1991.
- [4] Kenneth P. Birman, "The process group approach to reliable distributed computing," Technical Report TR91-1216, Cornell University Computer Science Department, Ithaca, NY, July 1991.
- [5] A. Ricciardi and K. Birman, "Using process groups to implement failure detection in asynchronous environments," in *ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 341-353, August 1991. Also available as TR91-1188, Dept. of Computer Science, Cornell Univ.
- [6] S. B. Shukla, F. Pires and D. Raghuram, "Design Implementation and Performance of a Decentralized Group Membership Protocol for Asynchronous Environments Using Ordered Views," Technical Report NPS-EC-93-006, Naval Postgraduate School, Monterey, California.
- [7] Anita Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault Tolerance Under Unix," in *ACM Transactions on Computer Systems*, pages 1-24, February 1989.
- [8] Bernd Walter, "A Robust and Efficient Protocol for Checking the Availability of Remote Sites," in *Proceedings of the Sixth Workshop on Distributed Data Management and Computer Networks, Berkeley, California*, pages 45-68, February 1982.
- [9] Kenneth P. Birman, Robert Cooper and Barry Gleeson, "Design alternatives for process group membership and multicast," Technical Report TR91-1257 (revision of TR91-1185, Jan. 1991), December 1991.
- [10] Shridhar B. Shukla and Devalla Raghuram, "Group Membership in Asynchronous Distributed Environments Using Logically Ordered Views," Technical Report NPS-EC-92-009, Naval Postgraduate School, Monterey, California, September 1992.

- [11] F. Jahanian and W. Moran Jr., "Strong, weak and hybrid group membership," in *Proceedings of the Second Workshop on the Management of Replicated Data*, Monterey, California, pages 34-38, November 1992. Also available as Technical Report RC 18040 (79173) 5/28/92, IBM Research Division, T. J. Watson Research Center, 1992.
- [12] L. Peterson, N. Bucholdz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," in *ACM Transactions on Computer Systems*, Montreal, Quebec, Canada, August 1989.
- [13] M. J. Fisher, N. A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus With One Faulty Process," in *Journal of the Association for Computing Machinery*, pages 374-382, 1985.
- [14] Paulo Verissimo, Jose Alves Marques, "Reliable Broadcast for Fault-Tolerance on Local Computer Networks," in *Proceedings IEEE Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama, pages 54-63, October 1990.
- [15] S. A. Bruso, "A Failure Detection And Notification Protocol for Distributed Computing Systems," in *Proceedings IEEE Conference on Distributed Computing Systems*, pages 116-123, 1985.
- [16] L. E. Moser, P. M. Melliar Smith, and V. Agrawala, "Membership Algorithm For Asynchronous Distributed Systems," in *Proceedings of the Eleventh International Symposium on Distributed Computing Systems*, 1991.
- [17] J.-M. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocol," in *ACM Transactions on Computer Systems*, pages 251-273, 1984.
- [18] K. P. Birman and T. A. Joseph, "Reliable Communications in the Presence of Failures," in *ACM Transactions on Computer Systems*, pages 47-76, August 1987.
- [19] Jean Walrand, *Communication Networks: A First Course*, Aksen Associates, 1991.
- [20] W. Richard Stevens, *Unix Network Programming*, Prentice Hall, 1990.

APPENDIX

```

/*****
 * GROUP MEMBERSHIP PROTOCOL - SIMPLE APPLICATION
 *
 *
 *   This is an example application, that creates an instance of the
 * membership protocol, and receives from it the most current group
 * view, when this changes.
 *   A mechanism for requesting the element shut-down is also
 * provided. This is accomplished with the following call:
 *
 *       kill(0, SIGALRM);
 *
 *   The application receives notice that the element has departed
 * by catching this signal (sent by mainproc) at function killelmnt()
 *
 *****/
 * Written by: Fernando J. Pires
 * Last revision: 9 Mar 1993
 *
 *****/

#include "gmp.h"
#include "socutil.c"

void killelmnt();

void main(argc, argv)
int  argc;
char *argv[];
{
    int          apIsocun, newsoc, childpid, clen, msglen;
    char         *apIpath, *groupname, *sitelist, *msg;
    struct sockaddr_uncaller_addr;
```

```

/*****
    DETERMINE INPUT ARGUMENTS (command line)
*****/

switch(argc){
    case 1:groupname = "group0";
        sitelist = (char*) NULL;
        break;
    case 2:groupname = argv[1];
        sitelist = (char*) NULL;
        break;
    case 3:groupname = argv[1];
        sitelist = argv[2];
        break;
    default: printf("usage: simpleapp [groupname [sitelist ]]\n");
        exit(-1);
}

/*****
    OPEN LOCAL SOCKET (where group views are received)
*****/

aplpath = UNIXSTR_TMPL;
mktemp(aplpath);
aplsocun = createUN(aplpath);
listen(aplsocun,5);

/*****
    ESTABLISH A SIGNAL HANDLER TO INTERCEPT
    THE ELEMENT FAILURE SIGNAL FROM mainproc
*****/

signal(SIGALRM, killelmt);

```

```

/*****
EXECUTE GMP's MAIN PROCESS
*****/
if ( (childpid = fork()) == -1 )
    printf("Can't fork\n");
else if (childpid == 0){/* child process */
    execlp("mainproc", "mainproc", aplpath, groupname, sitelist, (char*)NULL);
    printf("Error executing mainproc\n");
    exit(1);}

/*****
EXECUTE LOOP TO RECEIVE UPDATED GROUP VIEWS
*****/
while ( (newsoc = accept(aplsocun, (struct sockaddr*) &caller_addr, &crlen)) >= 0 ){
    if((msglen=recmsg(newsoc, &msg)) < 0){
        printf("APPLICATION: read error\n");
        break;
    }
    msg[msglen]=NULL; /*turn message into string */
    printf("#####\n");
    printf("*****\n");
    printf("APPLICATION: received group view => ||%s|\n",msg);
    printf("*****\n");
    printf("#####\n");
    free(msg);
    close(newsoc);
}
printf("APPLICATION: accept error\n");

/* Send signal to mainproc and to itself, requesting element shut-down */
kill(0, SIGALRM);
}

/*****
SIGNAL HANDLER THAT CATCHES ELEMENT DEPARTURE
*****/

```

```
void killelmt()
{
    printf("APPLICATION: mainproc has returned\n");

    /* Terminate all running processes */
    sleep(2); /* Allow time for mainproc to shut-down */
    kill(0, SIGKILL);
}
```

```

/*****
* GROUP MEMBERSHIP PROTOCOL - MAIN PROCESS
*
*   This program is executed by the application, and spawns a
* complete implementation of an element running the Membership Protocol
*   This program waits for a child to cease execution (meaning
* that the element has or is to cease existence, and then releases all
* resources to the operating system.
*   It also catches signals from the application requesting the
* element shut-down.
*
*   Example of code used by the application to run this program:
*
* //create unix socket where GroupViews are to be received//
* char*socpath;
* int socfd;
*
* socpath = UNIXSTR_TMPL;// Default path template //
* mktemp(socpath);// Get unique file name //
* socfd = createUN(socpath);// Create unix socket //
* listen(socfd,5);
*
* //fork and execute mainproc//
* if ( (childpid = fork()) == -1 )
*     printf("Can't fork\n");
* else if (childpid == 0){// child process //
*     execlp("mainproc", "mainproc", socpath, grouppathname, sitelist,
* (char*)NULL);
*     printf("Error executing front\n");
*     exit(1);}
*
*   Notes: sitelist is a string with host names, separated by
*         '=' characters. Example: "sun2=sun10=aditya=taurus"
*         This list can be empty, in which case the local host
*
*         Gruopname is optional. If no argument is provided
*         it defaults to 'group0'.

```

```

*
*****
* Written by: Fernando J. Pires
* Last revision: 9 Mar 1993
*
*****/

```

```

#include "gmp.h"
#include "socutil.c"

```

```

void killelmt();
/* these variables are common to 'mainproc' and 'killelmt' */
char      *fpath, *bpath, *stmonpath, *streppath, *timerpath;
char      *joinppath, *intmbrpath, *agrppath, *comppath;
char      *gvmpath, *stmpath, *tpmpath, *aplp, *grouppathname;

```

```

int main(argc,argv)
int  argc;
char *argv[];
{
    int      fsocudp, fsocun, bsocudp, bsocun;
    int      stmonfd, strepfd, timerfd, joinpfd, intmbrfd;
    int      agrpfd, compfd, gvmfd, stmfd, tpmfd;
    char      sfudp[MAXFD], sbudp[MAXFD], sfun[MAXFD], sbun[MAXFD];
    char      sstmon[MAXFD], sstrep[MAXFD], stimer[MAXFD];
    char      sjoinp[MAXFD], sintmbr[MAXFD], sagrp[MAXFD];
    char      scomp[MAXFD], sgvm[MAXFD], sstm[MAXFD], stpm[MAXFD];
    u_short   fport, bport;
    char      sfport[MAXPORT], sbport[MAXPORT];
    int      childpid;
    char      my_name[MAXHOSTNAME+1], *ip_addr;
    char      my_addr[MAXLMTSIZE+1];
    char      *groupname, *sitelist;
    struct hostent* hptr;

```

```

/*****
ESTABLISH A SIGNAL HANDLER TO INTERCEPT
THE ELEMENT FAILURE SIGNAL FROM APPLICATION
*****/

signal(SIGALRM, killelmt);

/*****
DETERMINE ADDRESS OF CURRENT ELEMENT
*****/

if (gethostname(my_name, MAXHOSTNAME) == 0)
    printf("My name is %s\n", my_name);
else
    printf("gethostname error\n");
if (hptr=gethostbyname(my_name))
    printf("Success, found %s , also known as %s\n",
        hptr->h_name, hptr->h_aliases[0]);
else
    printf("Host %s not found\n", my_name);

ip_addr = (char*) inet_ntoa( *((struct in_addr*)( hptr->h_addr)));
printf("My IP address is %s\n", ip_addr);

/*****
DETERMINE INPUT ARGUMENTS (command line)
*****/

switch(argc){
    case 2: aplpath = argv[1];
        groupname = "group0";
        sitelist = my_name;
        break;
    case 3: aplpath = argv[1];
        groupname = argv[2];
        sitelist = my_name;
        break;
    case 4: aplpath = argv[1];
        groupname = argv[2];

```

```

        sitelist = argv[3];
        break;
    default: printf("usage: mainproc aplsoc [groupname [sitelist ]]\n");
        exit(-1);
    }

    printf("MAINPROC: start execution\n");
    printf("MAINPROC: aplsoc = %s, groupname = %s, sitelist = %s\n", aplpath, groupname,
    sitelist);
    grouppathname = CALLOC(strlen(groupname)+6, char);
    strcpy(grouppathname, "/tmp/");
    strcat(grouppathname, groupname);

    /******
    OPEN SOCKETS FOR ALL PROCESSES
    *****/

    /* open FRONT port UDP socket ( an Internet Datagram Socket) */
    fport = htons(0);
    fsocudp = createUDP(&fport); /* ask for an available port */
    sprintf(sfudp, "%d", fsocudp);
    printf("FRONT: internet port => %d\n", ntohs(fport));
    sprintf(sfport, "%d", fport);
    printf("\n");

    /* open a FRONT port Unix Domain Stream socket */
    fpath = UNIXSTR_TMPL; /* Default path template */
    mktemp(fpath); /* Get unique file name */
    fsocun = createUN(fpath);
    listen(fsocun, 5);
    sprintf(sfun, "%d", fsocun);
    printf("FRONT: unix socket path => %s\n", fpath);
    printf("\n");

    /* open BACK port UDP socket ( an Internet Datagram Socket) */
    bport = htons(0);
    bsocudp = createUDP(&bport); /* ask for an available port */
    sprintf(sbudp, "%d", bsocudp);

```

```

printf("BACK: internet port => %d\n",ntohs(bport));
sprintf(sbport,"%d",bport);
printf("\n");

```

```

/* open a BACK port Unix Domain Stream socket */

```

```

    bpath = UNIXSTR_TMPL; /* Default path template */
    mktemp(bpath); /* Get unique file name */
    bsocun = createUN(bpath);
    listen(bsocun,5);
    sprintf(sbun,"%d",bsocun);
    printf("BACK: unix socket path => %s\n",bpath);
    printf("\n");

```

```

/* open a STATUS MONITOR Unix Domain Stream socket */

```

```

    stmonpath = UNIXSTR_TMPL; /* Default path template */
    mktemp(stmonpath); /* Get unique file name */
    stmonfd = createUN(stmonpath);
    listen(stmonfd,5);
    sprintf(sstmon,"%d",stmonfd);
    printf("STATUS MONITOR: unix socket path => %s\n",stmonpath);
    printf("\n");

```

```

/* open a STATUS REPORTER Unix Domain Stream socket */

```

```

    streppath = UNIXSTR_TMPL; /* Default path template */
    mktemp(streppath); /* Get unique file name */
    strepfd = createUN(streppath);
    listen(strepfd,5);
    sprintf(sstrep,"%d",strepfd);
    printf("STATUS REPORTER: unix socket path => %s\n",streppath);
    printf("\n");

```

```

/* open a TIMER Unix Domain Stream socket */

```

```

    timerpath = UNIXSTR_TMPL; /* Default path template */
    mktemp(timerpath); /* Get unique file name */
    timerfd = createUN(timerpath);
    listen(timerfd,5);
    sprintf(stimer,"%d",timerfd);
    printf("TIMER: unix socket path => %s\n",timerpath);

```

```

    printf("\n");

/* open a JOIN PROCESSOR Unix Domain Stream socket */
    joinppath = UNIXSTR_TMPL; /* Default path template */
    mktemp(joinppath); /* Get unique file name */
    joinpfd = createUN(joinppath);
    listen(joinpfd,5);
    sprintf(sjoinp,"%d",joinpfd);
    printf("JOIN PROCESSOR: unix socket path => %s\n",joinppath);
    printf("\n");

/* open a INTEGRATE MEMBER Unix Domain Stream socket */
    intmbrpath = UNIXSTR_TMPL; /* Default path template */
    mktemp(intmbrpath); /* Get unique file name */
    intmbrfd = createUN(intmbrpath);
    listen(intmbrfd,5);
    sprintf(sintmbr,"%d",intmbrfd);
    printf("INTEGRATE MEMBER: unix socket path => %s\n",intmbrpath);
    printf("\n");

/* open a AGREEMENT PROCESSOR Unix Domain Stream socket */
    agrppath = UNIXSTR_TMPL; /* Default path template */
    mktemp(agrppath); /* Get unique file name */
    agrpfd = createUN(agrppath);
    listen(agrpf,5);
    sprintf(sagrp,"%d",agrpfd);
    printf("AGREEMENT PROCESSOR: unix socket path => %s\n",agrppath);
    printf("\n");

/* open a COMMIT PROCESSOR Unix Domain Stream socket */
    comppath = UNIXSTR_TMPL; /* Default path template */
    mktemp(comppath); /* Get unique file name */
    compfd = createUN(comppath);
    listen(compfd,5);
    sprintf(scomp,"%d",compfd);
    printf("COMMIT PROCESSOR: unix socket path => %s\n",comppath);
    printf("\n");

```

```

/* open a GROUP VIEW MANAGER Unix Domain Stream socket */
gvmpath = UNIXSTR_TMPL; /* Default path template */
mktemp(gvmpath); /* Get unique file name */
gvmfd = createUN(gvmpath);
listen(gvmfd,5);
sprintf(sgvn,"%d",gvmfd);
printf("GROUP VIEW MANAGER: unix socket path => %s\n",gvmpath);
printf("\n");

/* open a STATUS TABLE MANAGER Unix Domain Stream socket */
stmpath = UNIXSTR_TMPL; /* Default path template */
mktemp(stmpath); /* Get unique file name */
stmfd = createUN(stmpath);
listen(stmfd,5);
sprintf(ssnm,"%d",stmfd);
printf("STATUS TABLE MANAGER: unix socket path => %s\n",stmpath);
printf("\n");

/* open a TOKEN POOL MANAGER Unix Domain Stream socket */
tpmpath = UNIXSTR_TMPL; /* Default path template */
mktemp(tpmpath); /* Get unique file name */
tpmfd = createUN(tpmpath);
listen(tpmfd,5);
sprintf(stpm,"%d",tpmfd);
printf("TOKEN POOL MANAGER: unix socket path => %s\n",tpmpath);
printf("\n");

/*****
DETERMINE COMPLETE ADDRESS OF CURRENT ELEMENT
*****/
strcpy(my_addr, ip_addr);
strcat(my_addr, ";");
strcat(my_addr, sfport);
strcat(my_addr, ";");
strcat(my_addr, sbport);
printf("My element address is ||%s||\n",my_addr);
printf("\n*****\n\n");

```

```

/*****
CREATE ALL PROCESSES
*****/

/* execute FRONT process */
if ( (childpid = fork()) == -1 )
    printf("Can't fork\n");
else if (childpid == 0){/* child process */
    execlp("front", "front", sfudp, sfun, my_addr, streppath, joinppath, intmbrpath,
(char*)NULL);
    printf("Error executing front\n");
    exit(1);}

/* execute BACK process */
if ( (childpid = fork()) == -1 )
    printf("Can't fork\n");
else if (childpid == 0){/* child process */
    execlp("back", "back", sbudp, sbun, my_addr, stmonpath, agrppath, (char*)NULL);
    printf("Error executing back\n");
    exit(1);}

/* execute GROUP VIEW MANAGER process */
if ( (childpid = fork()) == -1 )
    printf("Can't fork\n");
else if (childpid == 0){/* child process */
    execlp("gvm", "gvm", sgvm, my_addr, aplpath, grouppathname, (char*) NULL);
    printf("Error executing gvm\n");
    exit(1);}

/* execute STATUS TABLE MANAGER process */
if ( (childpid = fork()) == -1 )
    printf("Can't fork\n");
else if (childpid == 0){/* child process */
    execlp("stm", "stm", sstm, (char*) NULL);
    printf("Error executing stm\n");
    exit(1);}

```

```

/* execute TOKEN POOL MANAGER process */
    if ( (childpid = fork()) == -1 )
        printf("Can't fork\n");
    else if (childpid == 0){/* child process */
        execlp("tpm", "tpm", stpm, (char*) NULL);
        printf("Error executing tpm\n");
        exit(1);}

/* execute TIMER process */
    if ( (childpid = fork()) == -1 )
        printf("Can't fork\n");
    else if (childpid == 0){/* child process */
        execlp("timer", "timer", stimer, stmonpath, (char*) NULL);
        printf("Error executing timer\n");
        exit(1);}

/* execute STATUS REPORTER process */
    if ( (childpid = fork()) == -1 )
        printf("Can't fork\n");
    else if (childpid == 0){/* child process */
        execlp("strep", "strep", sstrep, my_addr, tmpath, fpath, (char*) NULL);
        printf("Error executing strep\n");
        exit(1);}

/* execute STATUS MONITOR process */
    if ( (childpid = fork()) == -1 )
        printf("Can't fork\n");
    else if (childpid == 0){/* child process */
        execlp("stmon", "stmon", sstmon, my_addr, stmpath, gvmpath, agrppath, timerpath,
bpath, (char*) NULL);
        printf("Error executing stmon\n");
        exit(1);}

/* execute JOIN PROCESSOR process */
    if ( (childpid = fork()) == -1 )
        printf("Can't fork\n");
    else if (childpid == 0){/* child process */

```

```

        execlp("joinp", "joinp", sjoinp, my_addr, stmpath, gvmpath, agrppath, intmbrpath,
bpath, fpath, grouppathname, sitelist, (char*) NULL);
        printf("Error executing joinp\n");
        exit(1);}

```

```

/* execute INTEGRATE MEMBER process */

```

```

        if ( (childpid = fork()) == -1 )
            printf("Can't fork\n");
        else if (childpid == 0){/* child process */
            execlp("intmbr", "intmbr", sintmbr, my_addr, gvmpath, stmpath, tmpath, bpath,
(char*) NULL);
            printf("Error executing intmbr\n");
            exit(1);}

```

```

/* execute AGREEMENT PROCESSOR process */

```

```

        if ( (childpid = fork()) == -1 )
            printf("Can't fork\n");
        else if (childpid == 0){/* child process */
            execlp("agrp", "agrp", sagrp, (char*) NULL);
            printf("Error executing agrp\n");
            exit(1);}

```

```

/* execute COMMIT PROCESSOR process */

```

```

        if ( (childpid = fork()) == -1 )
            printf("Can't fork\n");
        else if (childpid == 0){/* child process */
            execlp("comp", "comp", scomp, (char*) NULL);
            printf("Error executing comp\n");
            exit(1);}

```

```

/* close all open files */

```

```

        close(fsocudp);
        close(bsocudp);
        close(fsocun);
        close(bsocun);
        close(stmonfd);
        close(strepfd);

```

```

close(timerfd);
close(joinpfd);
close(intmbrfd);
close(agrpfdf);
close(compfd);
close(gvmfd);
close(stmfd);
close(tpmfd);

wait( (int*) NULL ); /* wait until one process exits */

sleep(80);
printf("MAIN PROCESS: One child has returned\n");

/* Remove Unix Domain socket links */
unlink(fpath);
unlink(bpath);
unlink(stmonpath);
unlink(streppath);
unlink(timerpath);
unlink(joinppath);
unlink(intmbrpath);
unlink(agrppath);
unlink(comppath);
unlink(gvmppath);
unlink(stmpath);
unlink(tpmpath);
unlink(aplpath);

/* Signal the application that the element has ceased existence */
kill(0, SIGALRM);
}

/*****
SIGNAL HANDLER THAT CATCHES ELEMENT DEPARTURE
*****/

```

```

void killelmt()
{
    printf("MAINPROC: application has requested shut-down\n");

    /* Remove Unix Domain socket links */
    unlink(fpath);
    unlink(bpath);
    unlink(stmonpath);
    unlink(streppath);
    unlink(timerpath);
    unlink(joinppath);
    unlink(intmbrpath);
    unlink(agrppath);
    unlink(comppath);
    unlink(gvmpath);
    unlink(stmpath);
    unlink(tpmpath);
    unlink(aplpath);
    remove(grouppathname);
    free(grouppathname);
}

```

```

/*****
* BACK PORT MANAGER
*
*   This program is executed by mainproc.
*
*****/

* Written by: Fernando J. Pires
* Last revision: 8 Mar 1993
*
*****/

#include "gmp.h"
#include "msgutil.c"
#include "socutil.c"
#include "fifoutil.c"

int main(argc,argv)
int  argc;
char *argv[];
{
    int          sockudp, sockun, newsoc;
    int          msgtype, clen, msglen;
    int          expectsrnbr = 0;
    char         *stmon, *agrp, *my_addr, *msg;
    char         *acwnbr, *target, *originator, *extmsg;
    struct sockaddr_ununcaller_addr;
    fd_set       fd_set;
    fdread;

    printf("BACK: start execution\n");

    if (argc == 6){
        sockudp = atoi(argv[1]);
        sockun = atoi(argv[2]);
        my_addr = argv[3];
        stmon = argv[4];
        agrp = argv[5];
    }
}

```

```

else{
    printf("Usage: BACK sockudp sockun my_addr stmon agrp\n");
    exit(1);
}

/* Initialize acwnbr with a null string */
acwnbr = CALLOC(1,char);

while(TRUE){
    FD_ZERO(&fdread);
    FD_SET(sockudp, &fdread);
    FD_SET(sockun, &fdread);

    /*
     * Wait for a connection from a client process, either at
     * the Internet or Unix socket.
     */

    if (select(32, &fdread, NULL, NULL, NULL) < 0){
        printf("BACK PORT: select error\n");
        exit(1);
    }

    if (FD_ISSET(sockun, &fdread)){/* Unix socket */
        clen = sizeof(uncaller_addr);
        if ( (newsoc = accept(sockun, (struct sockaddr*) &uncaller_addr, &clen)) < 0 ){
            printf("BACK unix: accept error\n");
            exit(1);
        }
        if((msglen=recmsg(newsoc, &msg)) < 0){
            printf(" BACK unix: read error\n");
            exit(1);
        }
        msg[msglen]=NULL; /*turn message into string */

        printf("BACK unix: received => ||%s||\n",msg);
    }
}

```

```

/* Determine which message was received */

msgtype = in_msg_type(msg);
switch (msgtype){
    case STATUSQRY: free(acwnbr);
                    acwnbr = get_target(msg);
printf("BACK unix: acwnbr = ||%s|\n", acwnbr);
    case INITPARAM:
    case JOINREQST: target = get_target(msg);
                    extmsg = int_2_ext(msg, 0, my_addr);
                    send_msg_front(extmsg, target);
                    free(target);
                    free(extmsg);
                    break;

    default: exit(1);
}
free(msg);
close(newsoc);
}

if (FD_ISSET(sockudp, &fdread)){/*Internet socket*/
    if ( (msglen=recmsg(sockudp,&msg)) < 0 ){
        printf("BACK internet: receive error\n");
        exit(1);
    }
    msg[msglen]=NULL; /*turn message into string */

    printf("BACK internet: received => ||%s|\n", msg);

    originator = get_originator(msg);
    if (strcmp(acwnbr, originator) == 0){
        /* accept msg only from acwnbr */

        /* Determines which message was received */

        msgtype = ext_msg_type(msg);

```

```

switch (msgtype){
    case STATUSRPT:send_msg_in(msg, stmon);
                    break;
    case TOKENTOKN:if(expectsrnr == get_sr_nbr(msg)){
                    send_msg_in(msg, agrp);
                    send_ack(msg, my_addr, acwnbr);
                    expectsrnr +=1;
                    }
                    break;
    case TOKENPOOL:send_msg_in(msg, agrp);
                    send_ack(msg, my_addr, acwnbr);
                    expectsrnr = get_sr_nbr(msg)+1;
                    break;

    default:exit(1);
}
}
free(originator);
free(msg);
}
}
}

```

```

/*****
* FRONT PORT MANAGER
*
*   This program is executed by mainproc.
*
*****/
* Written by:Fernando J. Pires
* Last revision:8 Mar 1993
*
*****/

```

```

#include "gmp.h"
#include "msgutil.c"
#include "socutil.c"
#include "fifoutil.c"

```

```

int main(argc,argv)
int  argc;
char *argv[];
{
    int          sockudp, sockun, newsoc;
    int          msgtype, clen, msglen;
    int          queue_counter=0, expectsrnr=0, srnr=0;
    char         *cwnbr, *topmsg, *msg, *extmsg;
    char         *my_addr, *strep, *joinp, *intmbr;
    struct sockaddr_ununcaller_addr;
    fd_set       fdread;
    queue        qu;
    queue        *msgqueue = &qu;

    printf("FRONT: start execution\n");

    if (argc == 7){
        sockudp = atoi(argv[1]);
        sockun = atoi(argv[2]);
        my_addr = argv[3];
        strep = argv[4];

```

```

        joinp = argv[5];
        intmbr = argv[6];
    }
    else{
        printf("Usage: front sockudp sockun my_addr strep joinp intmbr\n");
        exit(1);
    }

```

```

/* Initialize msgqueue */
qu.tail = qu.head = NULL;

```

```

/* Initialize cwnbr with a null string */
cwnbr = CALLOC(1,char);

```

```

while(TRUE){
    FD_ZERO(&fdread);
    FD_SET(sockudp, &fdread);
    FD_SET(sockun, &fdread);

    /*
     * Wait for a connection from a client process, either at
     * the Internet or Unix socket.
     */

    if (select(32, &fdread, NULL, NULL, NULL) < 0){
        printf("FRONT PORT: select error\n");
        exit(1);
    }

    if (FD_ISSET(sockun, &fdread)){/* Unix socket */
        clen = sizeof(uncaller_addr);
        if ( (newsoc = accept(sockun, (struct sockaddr*) &uncaller_addr, &clen)) < 0 ){
            printf("FRONT unix: accept error\n");
            exit(1);
        }
    }
}

```

```

if((msglen=readmsg(newsoc,&msg,"#")) < 0){
    printf(" FRONT unix: read error\n");
    exit(1);
}
msg[msglen]=NULL; /*turn message into string */

printf("FRONT unix: received => ||%s|\n",msg);

/* Determine which message was received */

msgtype = in_msg_type(msg);
switch (msgtype){
    case TOKENPOOL:flush_queue(msgqueue);
    case TOKENTOKN:extmsg = int_2_ext(msg, srnbr, my_addr);
                    enqueue(msgqueue,extmsg);
                    free(extmsg);
                    srnbr++;
                    queue_counter++;
                    break;
    case STATUSRPT:free(cwnbr);
                    cwnbr = get_target(msg);
                    extmsg = int_2_ext(msg, 0, my_addr);
                    send_msg_back(extmsg, cwnbr);
                    free(extmsg);
                    break;

    default:exit(1);
}
free(msg);

if (queue_counter != 0){
    get_queue_head(msgqueue,&topmsg);
    send_msg_back(topmsg, cwnbr);
    expectsrnbr = get_sr_nbr(topmsg);
    free(topmsg);
}

close(newsoc);

```

```

    }

    if (FD_ISSET(sockudp, &fdread))/*Internet socket*/{
        if ( (msglen=recmsg(sockudp, &msg)) < 0 ){
            printf("FRONT internet: receive error\n");
            exit(1);
        }
        msg[msglen]=NULL; /*turn message into string */

        printf("FRONT internet: received => ||%s||\n",msg);

        /* Determines which message was received */

        msgtype = ext_msg_type(msg);
        switch (msgtype){
            case STATUSQRY:send_msg_in(msg,src);
                break;
            case JOINREQST:
            case INITPARAM:send_msg_in(msg,joinp);
                break;
            case TOKENACKN:
                if (get_sr_nbr(msg) == expectsrnbr){
                    queue_counter--;
                    dequeue(msgqueue);
                }
                break;

            default:exit(1);
        }
        free(msg);
    }
}
}

```

```

/*****
* SOCKET INTERFACE AUXILIARY FUNCTIONS
*
* The following functions are available to be used:
*
* int createUDP(u_short port);
* int createUN(char *path);
* int connectUN(char *server_path);
* int readmsg(int fd, char **ptr, char *eom);
* int writemsg(int fd, char *ptr, int n);
* void senmsg(char *msg, int n, char *IPaddr, u_short port);
* int recmsg(int fd, char **str, );
*
* Refer to the function header comments for detailed info.
* Other functions in this file are used internally, and should
* not be used directly.
*
*****/
* Writen by:Fernando J. Pires
* Last revision: 18 Feb 1993
*
*****/

```

```

/*****
    createUDP - establish an UDP socket for a server
*****/
int createUDP(port)
u_short *port;

{
    struct sockaddr_in sin; /* Internet endpoint address */
    int sockfd; /* socket descriptor */
    int sinlen;

    bzero((char*)&sin, sizeof(sin)); /* clear address structure */
    sin.sin_family = AF_INET;

```

```

sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = *port;

/* Open the socket */

if ( ( sockfd = socket( PF_INET, SOCK_DGRAM, 0)) < 0){
    printf("\tcreateUDP: can't open internet socket \n");
    exit(1);}

/* Bind the socket */

sinlen = sizeof(sin);
if ( bind(sockfd, (struct sockaddr *) &sin, sinlen )< 0){
    printf("\tcreateUDP: can't bind local address \n");
    exit(1);}
if ( getsockname(sockfd, (struct sockaddr *) &sin, &sinlen )< 0){
    printf("\tcreateUDP: can't bind local address \n");
    exit(1);}
*port = sin.sin_port;

return sockfd;
}

/*****
createUN - establish an Unix Domain socket for a server
*****/

int createUN(path)
char* path;

{
    struct sockaddr_unsunx; /* Internet endpoint address */
    int sockfd; /* socket descriptor */
    int sunlen; /* Addr struct length */

    bzero((char*)&sunx,sizeof(sunx)); /* clear address structure */
    sunx.sun_family = AF_UNIX;
    strcpy(sunx.sun_path, path);
    sunlen = strlen(sunx.sun_path)+sizeof(sunx.sun_family);

```

```
/* Open the socket */
```

```
if ( ( sockfd = socket( AF_UNIX, SOCK_STREAM, 0)) < 0){  
    printf("\tcreateUN: can't open unix socket \n");  
    exit(1);}
```

```
/* Bind the socket */
```

```
unlink(path);/* in case it was left open by a previous call */  
if ( bind(sockfd, (struct sockaddr *) &sunx, sunlen )< 0){  
    printf("\tcreateUN: can't bind local path \n");  
    exit(1);}
```

```
return sockfd;
```

```
}
```

```
/******
```

```
connectUN - establish an Unix Domain socket for a client
```

```
******/
```

```
int connectUN(server_path)
```

```
char* server_path;
```

```
{
```

```
    struct sockaddr_unsunx;/* Internet endpoint address */
```

```
    int          sockfd;/* socket descriptor */
```

```
    int          sunlen;/* Addr struct length */
```

```
    bzero((char*)&sunx,sizeof(sunx));/* clear address structure */
```

```
    sunx.sun_family = AF_UNIX;
```

```
    strcpy(sunx.sun_path, server_path);
```

```
    sunlen = strlen(sunx.sun_path)+sizeof(sunx.sun_family);
```

```
/* Open the socket */
```

```
if ( ( sockfd = socket( AF_UNIX, SOCK_STREAM, 0)) < 0){  
    printf("\tconnectUN: can't open unix socket \n");
```

```

        exit(1);}

/* Connect to the server */

    if ( connect(sockfd, (struct sockaddr *) &sunx, sunlen )< 0){
        printf("\tconnectUN: can't connect to unix server \n");
        exit(1);}

    return sockfd;
}

/*****
    readn - Read "n" bytes from a descriptor
           Use in place of read() when fd is a stream socket
*****/
int readn(fd,ptr,nbytes)
register intfd;
register char*ptr;
register intnbytes;
{
    int    nleft, nread;

    nleft = nbytes;
    while (nleft > 0) {
        nread = read(fd,ptr,nleft);
        if ( nread < 0)
            return(nread);
        else if(nread == 0)
            break;

        nleft -= nread;
        ptr += nread;
    }
    return(nbytes-nleft);
}

```

```

/*****
    writen - Write "n" bytes to a descriptor
            Use in place of wite() when fd is a stream socket
*****/

```

```

int writen(fd,ptr, nbytes)
register intfd;
register char*ptr;
register intnbytes;
{
    int    nleft, nwriten;

    nleft = nbytes;
    while (nleft > 0) {
        nwriten = write(fd,ptr,nleft);
        if ( nwriten < 0)
            return(nwriten);
        nleft -= nwriten;
        ptr += nwriten;
    }
    return(nbytes-nleft);
}

```

```

/*****

```

readmsg - Read a complete message from a descriptor
 Use in place of read() when fd is a stream socket
 The message is assumed to be terminated by 'eom'.
 The function allocates the necessary space to build
 a non-Null terminated string '*str', plus space for
 an extra NULL (to be used by the calling function).
 The calling function must free the allocated space,
 when it is no longer necessary.
 Sample call sequence:

```

char *str;

len = readmsg(fd, &str, "#");
str[len] = NULL;

```

```

        ...
        free(str);
        *****/
int readmsg(fd, ptr, eom)
register intfd;
register char**ptr;
register char*eom;
{
    int  n, rc, maxlen;
    char c, msghead[HEADERSIZE+1], *tmp;

    /* get msg size */
    tmp = msghead;
    do {
        if ( (rc = read(fd, &c, 1)) != 1)
            return(0);
        *tmp++ = c;
    }while( c != '~');

    *--tmp = NULL; /* substitute NULL for '~' to end string */
    if ((maxlen = atoi(msghead)) == 0)
        return(0);
    /* allocate space for message and extra NULL*/
    *ptr = tmp = CALLOC(maxlen+1,char);

    /* get message character by character */
    for (n = 1; n <= maxlen; n++) {
        if ( (rc = read(fd, &c, 1)) == 1) {
            *tmp++ = c;
            if (c == *eom)
                break; /* End of message */
        } else if (rc == 0){
            if (n == 1){
                free(*ptr);
                return(0); /* EOF, no data read */
            } else
                break; /* EOF, data was read */
            /* Note: the calling function has to free

```

```

                                the allocated space */
        } else{
            free(*ptr);
            return(-1); /* error */
        }
    }
    return(n);
}

```

/******

writemsg - Writes a complete message 'ptr' of size 'n' to a file descriptor 'fd'. It appends an header that contains the size of the original message, plus a '~' as a separator. This header is to be processed by readmsg().
It returns the number of characters from the original message that were actually transmitted.
The original message is not changed.
Sample call:

```
n = writemsg(fd, str, strlen(str));
```

...

*****/

```

int writemsg(fd, ptr, n)
register intfd;
register char*ptr;
register intn;
{
    int  nwrite, len;
    char header[HEADERSIZE+1], *msg;

    if (n > (len = strlen(ptr)))
        n = len;
    sprintf(header, "%d~", n);
    msg = CALLOC(n+strlen(header)+1, char);
    strcpy(msg, header);
    strncat(msg, ptr, n);
}

```

```

nwrite = writen(fd,msg,strlen(msg));
free(msg);
return(nwrite-strlen(header));
}

```

```

/*****

```

senmsg - sends an external message 'msg' of size 'n'
 to the specified IP destination <IPaddr, port>.
 An header with the value of the size of the
 message, and a '~' as separator, is appended
 to the message. This header is to be processed
 by recmsg().
 The original message is not disturbed.
 Sample call:

```

    senmsg(msg, n, IPaddr, port);

```

```

*****/

```

```

void senmsg(msg, n, IPaddr, port)

```

```

char *msg;

```

```

int n;

```

```

char *IPaddr;

```

```

u_short port;

```

```

{

```

```

    struct sockaddr_in target_addr;

```

```

    struct hostent *phe;

```

```

    struct iovec iov[2];

```

```

    int sockfd, len;

```

```

    u_short outport;

```

```

    char header[HEADERSIZE];

```

```

    /* get the target UDP socket description */

```

```

    bzero((char*)&target_addr,sizeof(target_addr));/* clear address structure */

```

```

    target_addr.sin_family = AF_INET;

```

```

    target_addr.sin_port = port;

```

```

    if ((target_addr.sin_addr.s_addr = inet_addr(IPaddr)) == INADDR_NONE )

```

```

        if ( phe = gethostbyname(IPaddr) )

```

```

        bcopy(phe->h_addr, (char*)&target_addr.sin_addr, phe->h_length);
    else {
        printf("senmsg: can't get ||%s|| host entry\n", IPAddr);
        exit(1);
    }

    /* set a connection to the destination */
    outport = htons(0);
    sockfd = createUDP(&outport); /* request an arbitrary socket */
    connect(sockfd, (struct sockaddr*) &target_addr, sizeof(target_addr));

    /* assemble a scattered message including the msg size */
    if (n > (len = strlen(msg)))
        n = len;
    iov[0].iov_base = header;
    sprintf(header, "%d~", n);
    iov[0].iov_len = strlen(header);
    iov[1].iov_base = msg;
    iov[1].iov_len = n;

    /* send message */
    if ( writev(sockfd, &iov[0], 2) != (strlen(header)+n) ){
        printf("senmsg: write error on socket\n");
        exit(1);}

    close(sockfd);
}

```

/*****

recmsg - reads an external message 'msg' at the
 specified IP socket.

The message is atomically received, and is striped
 of the header (as created by senmsg()).

The function allocates the necessary space to build
 a non-Null terminated string '*str', plus space for
 an extra NULL (to be used by the calling function).

The calling function must free the allocated space,

when it is no longer necessary.

The function returns the message size, and -1 if an invalid message is received.

Sample call sequence:

```
char *str;

len = recmsg(fd, &str);
str[len] = NULL;
...
free(str);
*****/
int recmsg(fd, str)
register intfd;
register char**str;
{
    char *msghead, *tmp, *msgbuf, buf[HEADERSIZE+1];
    int msglen, recvlen, mlen;

    /* retrieve the header (the message is not removed) */
    if ((msglen = recv(fd, buf, HEADERSIZE, MSG_PEEK)) < 0 ){
        printf("recmsg: header error\n");
        exit(1);
    }
    buf[HEADERSIZE]=NULL;
    msghead = strtok(buf, "~");
    if ((msglen = atoi(msghead)) == 0)
        return(0);
    /* allocate space for entire message */
    msgbuf = CALLOC(msglen + HEADERSIZE + 1, char);

    /* get entire message */
    if ((recvlen = recv(fd, msgbuf, msglen+HEADERSIZE, 0)) < 0 ){
        printf("recmsg: message error\n");
        free(msgbuf);
        return(-1);
    }
    msgbuf[recvlen]=NULL;
```

```
/* extract message info and discard header */  
tmp = strtok(msgbuf,"~");  
tmp = strtok(NULL,"~");  
mlen = strlen(tmp);  
*str = CALLOC(mlen + 1, char);  
strcpy(*str,tmp);  
free(msgbuf);  
return(mlen);  
}
```

```

/*****
* MESSAGE HANDLING AUXILIARY FUNCTIONS
*
* The following functions are available to be used:
*
* link *str2list(char *str, char *token);
* char *list2str(link *list, char *header, char* htok, char *ltok);
* void removelist(link *list);
* int listsize(link *list);
* int getfromlist(link *list, char **str, int n);
* char *int_2_ext(char *inmsg, int srnbr, char *orig);
* int get_sr_nbr(char *msg);
* int in_msg_type(char *inmsg);
* int ext_msg_type(char *extmsg);
* char *get_target(char *str);
* char *get_originator(char *str);
*
* Refer to the function header comments for detailed info.
* Other functions in this file are used internally, and should
* not be used directly.
*
*****/

* Written by:Fernando J. Pires
* Last revision:8 Mar 1993
*
*****/

```

```

struct link{
    char*data;
    struct link*next;
};
typedef struct linklink;

```

```

/*****
    str2list - parse a string, creating a list of nodes, each
               of which points to a field of the original string.
*****/

```

The fields are originally separated by token.
 In the original string, tokens are replaced by NULL
 After the list is no longer needed, removelist()
 must be called for garbage collection.

```

*****/
link* str2list(str, token)
char *str;
char *token;
{
    link *msglst, *tmp;
    char *ptr;

    tmp = msglst = CALLOC(1,link);

    tmp->data = ptr = strtok(str,token);
    while (ptr = strtok(NULL,token)){
        tmp->next = CALLOC(1,link);
        tmp = tmp->next;
        tmp->data = ptr;
    }
    return(msglst);
}

```

/******

list2str - assembles a string from a list generated by
 str2list() and appends it to the string 'header'.
 'htok' is inserted after the header. 'ltok' is
 inserted in between each new field and a NULL is
 added at the end.

Notes:

'header' has to be dynamically allocated
 (it cannot be static data). The best way to
 initialize it is to use "header = CALLOC(1,char);"

If the list is empty the header is returned
 without changes.

The resulting message has to be deallocated
 with a free() call when it is no longer needed.

*****/

```

char* list2str(list, header, htok, ltok)
link *list;
char *header;
char *htok;
char *ltok;
{
    int len = 0;
    link *ptr = list;

    if (list){
        while (ptr){ /* determine size of string to be used */
            len += strlen(ptr->data)+1; /* Reserve space for
                                         token and NULL */
            ptr = ptr->next;
        }
        header = REALLOC(header,strlen(header)+2+len,char);
        if (htok)
            strcat(header,htok); /* insert htok */
        if (len){
            while (list){ /* assemble the string */
                strcat(header,list->data);
                list = list->next;
                if (list && ltok) /*add ltok except after
                                   last field*/
                    strcat(header,ltok);
            }
        }
        return(header);
    }
}

```

removelist - Deallocates the space used by str2list() to
generate a list. This function must be called for
every list, once it is no longer needed.

*****/

```

void removelist(list)
link *list;

```

```

{
    if (list->next)
        removelist(list->next);
        free(list);
}

```

```

/*****
    getfromlist - Get the nth field from list. Upon execution 'str'
                  points to the nth field.
                  Returns n if the call is successful, or zero if the
                  list has less than n fields.
*****/

```

```

int getfromlist(list, str, n)
link *list;
char **str;
int n;
{
    int p;

    if (list == NULL)
        return(0);

    if (n == 1){
        *str = list->data;
        return(n);
    }else{
        p = getfromlist(list->next, str, n-1);
        if (p)
            return(n);
        else
            return(0);
    }
}

```

```

/*****
    listsize - Return the number of elements of a list

```

Returns n > 0 for a non-empty list, and 0 otherwise.

*****/

```
int listsize(list)
```

```
link *list;
```

```
{
```

```
    int    n = 0;
```

```
    while (list){
```

```
        list = list->next;
```

```
        n++;
```

```
    }
```

```
    return(n);
```

```
}
```

*****/

int_2_ext - convert 'inmsg' into an external message.

'srnbr' is converted to a string and is used as
a prefix to 'inmsg'. 'orig' is the address of the
local element and it is inserted after 'snrb'.

An NL character is used to separate the fields.

The original 'inmsg' is not modified.

To convert an internal message to external format use:

```
extmsg = int_2_ext(inmsg, srnbr, orig);
```

```
...
```

```
free(extmsg);
```

If the serial number is not relevant set 'snbr' to 0.

*****/

```
char *int_2_ext(inmsg, srnbr, orig)
```

```
char *inmsg;
```

```
int srnbr;
```

```
char *orig;
```

```
{
```

```
    int  msgsize;
```

```
    char temp[HEADERSIZE];
```

```
    char *extmsg;
```

```

    msgsize = strlen(inmsg);
    sprintf(temp,"%d\n",srnbr);
    extmsg = CALLOC(strlen(temp)+strlen(orig)+msgsize+2, char);
    strcpy(extmsg,temp);
    strcat(extmsg,orig);
    strcat(extmsg,"\n");
    strcat(extmsg,inmsg);
    return(extmsg);
}

```

get_sr_nbr - extracts the serial number of a message that
 was previously retrieved from the queue, or
 received at the external port.
 The original message is not disturbed.
 The function returns the serial number, or -1
 if an error occurs.

*****/

```

int get_sr_nbr(msg)
char *msg;
{
    char *tmp, *srnbrstr;
    link *list;

    /* make a copy of the message */
    tmp = CALLOC(strlen(msg)+1, char);
    strcpy(tmp, msg);

    /* break the message into a list of fields */
    list = str2list(tmp, "\n");
    if (getfromlist(list, &srnbrstr, 1) != 1){ /* get 1st field */
        printf("get_sr_nbr error\n");
        removelink(list);
        free(tmp);
        return(-1);
    }
}

```

```

    }
    removelist(list);
    free(tmp);
    return(atoi(srnbrstr));
}

```

```

/*****

```

msg_type - returns the integer value corresponding to
the type of the string 'type', as defined in gmp.h

```

*****/

```

```

int msg_type(type)
char *type;
{
    if (strcmp(type,"tokentkn")==0)
        return(TOKENTKN);
    else if (strcmp(type,"tokenpool")==0)
        return(TOKENPOOL);
    else if (strcmp(type,"tokenackn")==0)
        return(TOKENACKN);
    else if (strcmp(type,"statusqry")==0)
        return(STATUSQRY);
    else if (strcmp(type,"statusrpt")==0)
        return(STATUSRPT);
    else if (strcmp(type,"statustbl")==0)
        return(STATUSTBL);
    else if (strcmp(type,"initparam")==0)
        return(INITPARAM);
    else if (strcmp(type,"joinreqst")==0)
        return(JOINREQST);
    else if (strcmp(type,"updstatus")==0)
        return(UPDSTATUS);
    else if (strcmp(type,"groupview")==0)
        return(GROUPVIEW);
    else if (strcmp(type,"updatview")==0)
        return(UPDATVIEW);
    else if (strcmp(type,"sndinipar")==0)
        return(SNDINIPAR);
}

```

```

    else if (strcmp(type,"inittoken")==0)
        return(INITTOKEN);
    else if (strcmp(type,"viewreqst")==0)
        return(VIEWREQST);
    else if (strcmp(type,"statreqst")==0)
        return(STATREQST);
    else if (strcmp(type,"tokpreqst")==0)
        return(TOKPREQST);
    else if (strcmp(type,"initgview")==0)
        return(INITGVIEW);
    else if (strcmp(type,"inittable")==0)
        return(INITTABLE);
    else if (strcmp(type,"inittpool")==0)
        return(INITTPOOL);
    else if (strcmp(type,"timeout__")==0)
        return(TIMEOUT__);
    else if (strcmp(type,"starttimr")==0)
        return(STARTTIMR);
    else if (strcmp(type,"delttoken")==0)
        return(DELTOKEN);
    else
        return(INVALDM$SG);
}

```

```

/*****

```

in_msg_type - extracts the type field of a message that
 was previously received at the internal port.
 The original message is not disturbed.
 The function returns an integer whose value is
 defined in 'gmp.h', or -1 if an error occurs.

```

*****/

```

```

int in_msg_type(inmsg)
char *inmsg;
{
    int msgtype;
    char *tmp, *type;

```

```

link *list;

/* make a copy of the message */
tmp = CALLOC(strlen(inmsg)+1, char);
strcpy(tmp, inmsg);

/* break the message into a list of fields */
list = str2list(tmp, "\n#");
if (getfromlist(list, &type, 1) != 1){ /* get 1st field */
    printf("in_msg_type error\n");
    removelist(list);
    free(tmp);
    return(-1);
}
msgtype = msg_type(type);
removelist(list);
free(tmp);
return(msgtype);
}

```

/**

ext_msg_type - extracts the type field of a message that
 was previously received at the external port.
 The original message is not disturbed.
 The function returns an integer whose value is
 defined in 'gmp.h', or -1 if an error occurs.

```

int ext_msg_type(extmsg)
char *extmsg;
{
    int  msgtype;
    char *tmp, *type;
    link *list;

    /* make a copy of the message */

```

```

    tmp = CALLOC(strlen(extmsg)+1, char);
    strcpy(tmp, extmsg);

    /* break the message into a list of fields */
    list = str2list(tmp, "\n#");
    if (getfromlist(list, &type, 3) != 3){ /* get 2nd field */
        printf("ext_msg_type error\n");
        removelist(list);
        free(tmp);
        return(-1);
    }
    msgtype = msg_type(type);
    removelist(list);
    free(tmp);
    return(msgtype);
}

/*****
get_target - extracts the destination field of an
internal message.
The original message is not disturbed. The result
is stored on a dynamic array 'nbr', that has to
be deallocated before reusing.
Sample call:

    char *nbr;
    ...
    nbr = get_target(msg);
    ...
    free(nbr);
*****/
char *get_target(str)
char *str;
{
    char *nbr, *nbrtmp, *tmp;
    link *list;

```

```

    tmp = CALLOC(strlen(str)+1, char);
    strcpy(tmp,str);
    list = str2list(tmp, "\n ");
    if ( getfromlist(list, &nbrtmp, 2) != 2){
        printf("get_target error\n");exit(-1);}
    nbr = CALLOC(strlen(nbrtmp)+1, char);
    strcpy(nbr,nbrtmp);
    removelist(list);
    free(tmp);
    return (nbr);
}

```

get_originator - extracts the originator field from the header of an external message.
 The original message is not disturbed. The result is stored on a dynamic array 'nbr', that has to be deallocated before reusing.
 Sample call:

```

char *nbr;
...
nbr = get_originator(msg);
...
free(nbr);

```

*****/

```

char *get_originator(str)
char *str;
{
    char *nbr, *tmp;
    link *list;

    tmp = CALLOC(strlen(str)+1, char);
    strcpy(tmp,str);
    list = str2list(tmp, "\n ");
    if ( getfromlist(list, &nbr, 2) != 2){

```

```
        printf("get_originator error\n");exit(-1);}
removelist(list);
free(tmp);
return (nbr);
}
```

```

/*****
* QUEUE HANDLING AUXILIARY FUNCTIONS FOR FIFO PROCESSES
*
* The following functions are available to be used:
*
* void enqueue(queue *qptr, char *msg);
* void dequeue(queue *qptr);
* void get_queue_head(queue *qptr, char **msg);
* void flush_queue(queue *qptr);
* void send_ack(char *msg, char *orig, char *dest);
* void send_msg_back(char *msg, char *dest);
* void send_msg_front(char *msg, char *dest);
* void send_msg_in(char *msg, char *dest);
*
* Refer to the function header comments for detailed info.
* Some functions in this file need socutil.c and msgutil.c
*
*****/
* Written by: Fernando J. Pires
* Last revision: 1 Feb 1993
*
*****/

```

```

struct node{
    char *data;
    struct node *previous;
    struct node *next;
};
typedef struct node node;

struct queue{
    node *tail;
    node *head;
};
typedef struct queue queue;

```

```
/******
```

enqueue - inserts the external message 'msg' at the
tail of the queue 'quptr'.

The original 'msg' is not modified.

'quptr' must be created before the first call to
enqueue(). To create a queue use:

```
queue qu; / declaration /  
queue *quptr = &qu; / initialize pointer /  
qu.tail = qu.head = NULL; / empty queue /
```

```
enqueue(quptr, msg); / function call /
```

```
*****/
```

```
void enqueue(quptr, msg)
```

```
queue *quptr;
```

```
char *msg;
```

```
{
```

```
    node *ptlmnt;
```

```
    ptlmnt = CALLOC(1,node);
```

```
    ptlmnt->data = CALLOC(strlen(msg)+1,char);
```

```
    strcpy(ptlmnt->data, msg);
```

```
    if (quptr->tail == NULL){
```

```
        quptr->tail = ptlmnt;
```

```
        quptr->head = ptlmnt;
```

```
        quptr->tail->previous = NULL;
```

```
        quptr->tail->next = NULL;
```

```
    } else{
```

```
        quptr->tail->previous = ptlmnt;
```

```
        ptlmnt->next = quptr->tail;
```

```
        ptlmnt->previous = NULL;
```

```
        quptr->tail = ptlmnt;
```

```
    }
```

```
}
```

```
/******
```

dequeue - remove a msg from the head of the queue 'quptr'.

Sample call:

```
dequeue(quptr);
```

```
*****/
```

```
void dequeue(quptr)
```

```
queue *quptr;
```

```
{
```

```
    node *tmp;
```

```
    if (quptr->head != NULL){
```

```
        tmp = quptr->head;
```

```
        quptr->head = tmp->previous;
```

```
        if (quptr->head == NULL)
```

```
            quptr->tail = NULL;
```

```
        else
```

```
            quptr->head->next = NULL;
```

```
        free(tmp->data);
```

```
        free(tmp);
```

```
    }
```

```
}
```

```
/******
```

get_queue_head - returns a pointer to the message at
the head of the queue.

Sample call:

```
get_queue_head(quptr, &msg);
```

```
*****/
```

```
void get_queue_head(quptr, msg)
```

```
queue *quptr;
```

```
char **msg;
```

```
{
```

```
    if (quptr->head)
```

```
        *msg = quptr->head->data;
```

```

        else
            *msg = NULL;
    }

    /*****
    flush_queue - remove all nodes of 'quptr' from memory.
    All used memory is deallocated.
    'quptr' remains a valid empty queue and can be
    reused by enqueue().
    *****/
void flush_queue(quptr)
queue*quptr;
{
    node *tmp;

    while (quptr->head != NULL){
        tmp = quptr->head->previous;
        free(quptr->head->data);
        free(quptr->head);
        quptr->head = tmp;
    }
    quptr->tail = NULL;
}

    /*****
    send_msg_front - sends an external message 'msg' to the
    front port of the specified IP destination 'dest'.
    'dest' is a string with element address format.
    The original message is not disturbed.
    Sample call:

        send_msg_front(msg, dest);
    *****/
void send_msg_front(msg, dest)
char *msg, *dest;

```

```

{
    link *list;
    char *IPaddr, *frontport, *tmp;
    u_shortport;

    /* make a copy of the address */
    tmp = CALLOC(strlen(dest)+1,char);

    list = str2list(dest, ",");
    if ( getfromlist(list, &IPaddr, 1) != 1){
        printf("sen_msg_front: IP address error\n");exit(-1);}
    if ( getfromlist(list, &frontport, 2) != 2){
        printf("sen_msg_front: front port error\n");exit(-1);}
    port = htons( (u_short)atoi(frontport) ); /* convert port # to network format */

    senmsg(msg, strlen(msg), IPaddr, port);

    removelist(list);
    free(frontport);
    free(IPaddr);
    free(tmp);
}

```

/******

send_msg_back - sends an external message 'msg' to the
back port of the specified IP destination 'dest'.
'dest' is a string with element address format.
The original message is not disturbed.
Sample call:

```
send_msg_back(msg, dest);
```

*****/

```
void send_msg_back(msg, dest)
```

```
char *msg, *dest;
```

```
{
```

```
    link *list;
```

```
    char *IPaddr, *backport, *tmp;
```

```

    u_shortport;

    /* make a copy of the address */
    tmp = CALLOC(strlen(dest)+1,char);

    list = str2list(dest, ";");
    if ( getfromlist(list, &IPaddr, 1) != 1){
        printf("sen_msg_back: IP address error\n");exit(-1);}
    if ( getfromlist(list, &backport, 3) != 3){
        printf("sen_msg_back: back port error\n");exit(-1);}
    port = htons( (u_short)atoi(backport) ); /* convert port # to network format */

    senmsg(msg, strlen(msg), IPaddr, port);

    removelist(list);
    free(backport);
    free(IPaddr);
    free(tmp);
}

/*****
send_ack - assembles an ack as a reponse to an external
           message 'msg' and sends it to the
           front port of the specified IP destination 'dest'.
           'dest' is a string with element address format.
           The original message is not disturbed.
           Sample call:

           send_ack(msg, dest);
*****/
void send_ack(msg, orig, dest)
char *msg, *orig, *dest;
{
    link *list;
    char *ackmsg, *tmp;

    /* make a copy of the msg */

```

```

tmp = CALLOC(strlen(msg)+1,char);

/* assemble ack message */
list = str2list(tmp, "\n");
if ( getfromlist(list, &ackmsg, 1) != 1){
    printf("send_ack: serial number in error\n");exit(-1);}
ackmsg = REALLOC(ackmsg,strlen(ackmsg)+strlen(orig)+13,char);

    strcat(ackmsg,"\n");
    strcat(ackmsg,orig);
    strcat(ackmsg,"\ntokenackn#");

send_msg_front(ackmsg, dest);

removelist(list);
free(ackmsg);
free(tmp);
}

/*****
send_msg_in - sends an external message 'msg' to the
the specified unix socket destination 'dest'.
'dest' is a string with a path name.
The message is converted to internal format, before
transmission. The original message string is not
disturbed.
Sample call:

    send_msg_in(msg, dest);
*****/
void send_msg_in(msg, dest)
char *msg, *dest;
{
    link *list;
    int  msglen, sockfd;
    char *header, *tmp, *inmsg;

```

```

/* make a copy of the message */
msglen = strlen(msg);
tmp = CALLOC(msglen+1,char);
strcpy(tmp, msg);

/* discard external header */
list = str2list(tmp, "\n");
if ( getfromlist(list, &header, 3) != 3){
    printf("sen_msg_in: error\n");exit(-1);}
inmsg = msg + (header - tmp);

/* open and connect socket to server */
sockfd = connectUN(dest);

/* send msg to socket */
msglen = strlen(inmsg);
if ( (writemsg(sockfd, inmsg, msglen)) != msglen ){
    printf("send_msg_in: write error on socket\n");
    exit(1);}

removelist(list);
free(header);
free(tmp);
close(sockfd);
}

```

INITIAL DISTRIBUTION LIST

- | | |
|--|---|
| 1. Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145 | 2 |
| 2. Dudley Knox Library Code 52
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 3. Chairman Code EC
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 4. Professor Shridhar B. Shukla Code EC/Sh
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 5. Professor Amr Zaky Code CS/Za
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 6. Direcção do Serviço de Instrução e Treino
Administração Central de Marinha,
Praça do Comércio
1188 LISBOA CODEX
PORTUGAL | 4 |