(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION AND EVALUATION OF AN ASYNCHRONOUS GROUP MEMBERSHIP PROTOCOL

by

David J. Pezdirtz, Jr.

December, 1993

Thesis Advisor: Shridhar B. Shukla

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE Dec 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis, Final

4. TITLE AND SUBTITLE Implementation and Evaluation of an Asynchronous Group Membership Protocol	5. FUNDING NUMBERS
6. AUTHOR(S) David J. Pezdirtz, Jr.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES 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 U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT *(maximum 200 words)* A group membership protocol provides the mechanisms to ensure the consistent group views among a group-oriented distributed processes. The protocol is required to dynamically re-configure the group views among the various members in the event of a change to the group due to a new member joining or a member departing. The departure may be voluntary or involuntary. The protocol must provide a scheme to detect the failure of any of the members and re-configure the group. Multiple changes to the group must be perceived at all members in the same order. This thesis deals with a particular group membership protocol. The protocol structures the group as a logical ring. Changes to the group are accomplished using a two-phase scheme. The agreement phase consists of circulation of an *agree* token. Processing the token makes a pending change known to all members. The commit phase incorporates the changes in the correct order. This thesis presents an implementation of this asynchronous group membership protocol. The main feature is that the decentralized nature of the protocol eliminates the need for a dedicated coordinator of changes. The processing requirements for the protocol are likewise distributed. The processing time required to implement a change is explored.

14. SUBJECT TERMS Asynchronous Group Membership Protocol, Unix based, distributed processing.	15. NUMBER OF PAGES 196
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Implementation and Evaluation of an Asynchronous Group Membership Protocol

by

David J. Pezdirtz, Jr.
Lieutenant, United States Navy
B.S.C.S., University of Vermont, 1983

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1993

Author:

David J. Pezdirtz, Jr.

Approved by:

Shridhar B. Shukla, Thesis Advisor

Randy L. Borchardt, Second Reader

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

ABSTRACT

A group membership protocol provides the mechanisms to ensure the consistent group views among a group-oriented distributed processes. The protocol is required to dynamically re-configure the group views among the various members in the event of a change to the group due to a new member joining or a member departing. The departure may be voluntary or involuntary. The protocol must provide a scheme to detect the failure of any of the members and re-configure the group. Multiple changes to the group must be perceived at all members in the same order.

This thesis deals with a particular group membership protocol. The protocol structures the group as a logical ring. Changes to the group are accomplished using a two-phase scheme. The agreement phase consists of circulation of an ***agree*** token. Processing the token makes a pending change known to all members. The commit phase incorporates the changes in the correct order.

This thesis presents an implementation of this asynchronous group membership protocol. The main feature is that the decentralized nature of the protocol eliminates the need for a dedicated coordinator of changes. The processing requirements for the protocol are likewise distributed. The processing time required to implement a change to the group is shown to have a linear relationship to the group size.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

Even though this thesis bears my name, there are two people who contributed considerably to its completion . In addition to my efforts, the original code was written and developed by Prof. Shridhar Shukla and LT Fernando Pires, Portuguese Navy. Their help and cooperation made this project significantly smoother.

Finally, I would like thank my wife Sandi. Without her support and tolerance this thesis would not have been possible. To her, I dedicate this thesis.

I. INTRODUCTION

A. BACKGROUND

1. Distributed Computing

Distributed computing is at the forefront of today's computing research. The increased reliability and performance has resulted in distributed computing being used in various applications such as distributed control applications, distributed databases, and real-time settings [1]. The need for fault-tolerant systems is particularly important to military applications. Such applications typically require fault tolerant algorithms, real-time response, on-line reconfiguration, and other schemes to increase reliability. These requirements, however, lead to significant additional complexity. This complexity arises in part due to the reliable inter-processor communication required to implement the distributed processing. Networks are inherently unreliable and make reliable application level message passing a non-trivial task. One of the primary requirements of reliable distributed applications is a reliable multicast communication primitive. A group membership protocol simplifies the construction of a such a primitive [2].

2. Group Membership

Cooperating processes constituting a single application share resources and constitute a process group. Underlying the consistent behavior of such a group is the requirement that all members of the group have implicit knowledge of all other members in the group. Additionally, all members must perceive changes to the group in the same order. Group membership will change as processes join or depart the group. Departures may be voluntary or involuntary as in the case of a failure. An additional requirement is the timely detection of members failing. In order to enhance the robustness of the system, intra-member monitoring must occur. This can be a simple exchange of messages indicating that the process being monitored is still "live."

Historically, protocols solving the group membership problem have been of a centralized design. One member acted as the group host and had the responsibility of monitoring all subordinate members. All changes were detected by the host. Upon a change to the group view, the host broadcast the new group view to all members of the group. Obviously, problems can arise if the host itself fails. Voting must occur in order to elect a new host with the added overhead associated with the voting. Additionally, the processing requirements are unequally distributed between the members of the group as the host takes a major share of the load.

B. SCOPE AND CONTRIBUTION

In this thesis, an implementation of the decentralized asynchronous membership protocol is presented. The protocol was originally presented in [3] and [4]. It was further refined and partially implemented in [5]. This thesis presents a brief overview to the protocol. A more detailed explanation of the protocol and definitions can be found in [5]. This thesis covers additional refinements to the protocol necessary to correct flaws discovered in the coding and testing phase of development. Additionally, the performance of the protocol on an Ethernet Local Area Network (LAN) is presented and discussed.

C. ORGANIZATION OF THE THESIS

The thesis is divided into six chapters. Chapter II presents an overview of the basic operation of the protocol. Chapter III discusses the changes required to implement improvements to the protocol. Chapter IV covers possible further refinements to the protocol. In Chapter V, the performance of the current protocol is analyzed. Chapter VI presents the analysis and possible future areas of research. The code developed is included in the Appendix.

II. GROUP MEMBERSHIP PROTOCOL

In this chapter, the group membership protocol (GMP) is described. The original protocol was presented in [3] and further developed in [5]. This chapter presents an overview of the protocol.

A. GROUP MEMBERSHIP PROTOCOL OVERVIEW

1. Assumptions

The following assumptions are made by the GMP in order for proper implementation. A fully-connected network of reliable First-In / First-Out (FIFO) communication channels connecting operational members is assumed. All failures are assumed to be *crash* or *fail-stop*[6]. This implies that a message sent will be delivered unless the recipient has failed. However, there is no upper bound on the time of delivery.

Multiple changes to the membership are allowed simultaneously. However, the changes are committed one at a time and in the same order at all members.

A member is added to the group when a ***join*** is processed and is removed when a ***failure*** is perceived.

The group name is assumed to be public. Those elements that may wish to become members by joining the group have access to the group file which contains the current group view. A prospective member searches for the file on a given site. The group view is extracted and the prospective member sends the ***joinreqst*** to the appropriate address.

The protocol maintains three main databases at each member; the membership list (group view), status table and token pool. Each has a separate database manager to ensure mutual exclusion to all processes needing the services of the database.

2. Overview

The GMP guarantees that the group view changes occur in the same relative sequence at all operational members of the group.

The most significant feature of the GMP is the decentralization. No single member is responsible for detecting a change to the membership nor guaranteeing group view consistency among the group members. A logical ring is used to implement both of these functions in a distributed manner. The logical ring is a circular ordering of the members of the group.

The physical location of the member has no relation to the ordering of the logical ring. Within the ring structure the direction of traversal was arbitrarily chosen as clockwise. Given the structure each member only monitors its anti-clockwise neighbor, the ***acwnbr***. The ***acwnbr*** responds to a status query, ***statusqry***, with a status report, ***statusrpt***. Likewise, it sends a ***statusqry*** to its ***acwnbr***. Thus, every member monitors only one other member of the group and is itself monitored by it clockwise neighbor, ***cwnbr***.

Consider a five member group. Process p_0 was the initial member of the group. The other members joined in an order such that p_0 is the ***acwnbr*** of p_1, p_1 is the ***acwnbr*** of p_2, and so on. Member p_1 sends a ***statusqry*** to p_0, which responds with a ***statusrpt***. Likewise, p_2 sends a ***statusqry*** to p_1. This is illustrated in Figure 1. For clarity, only the monitoring and response is shown for the first set of neighbors.

The ring configuration changes as the group membership changes. The ring starts as a single member group with other members joining in some arbitrary order. All changes to the group are treated in a similar manner. Members wishing to join an existing group read the group membership file in the first active member located on the net. The joining member then sends a ***joinreqst*** to the first member of the group. If the initial parameters message, ***initparams***, is not received in a reasonable time, the request is transmitted to the next member in the group view file until all members have been attempted or a successful join has been completed.

A failure is considered an involuntary departure. When a member departs the group voluntarily, it simply stops responding to ***statusqrys***. It will then be perceived as failed and subsequently removed from the group. Delayed transmission of the ***statusrpt***, failure of the member to respond to the ***statusqry*** and a lost ***statusrpt*** will all result in the monitored member being detected as failed.

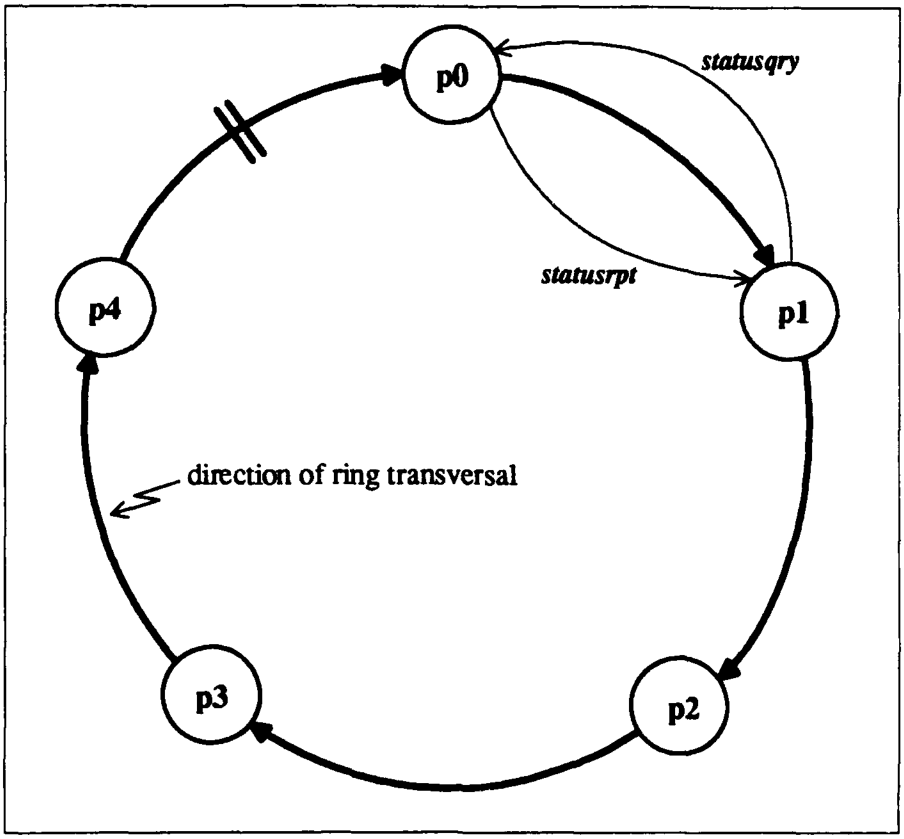

Figure 1 A Logical Ring

3. Processing of Individual Changes

The GMP allows for a two phase procedure for all changes to the group view, the ***agree*** phase and the ***commit*** phase. An ***agree*** token is circulated around the ring. Once the originator receives the token via the ring, all members have processed the ***agree*** token. At this point the ***agree*** token is converted to a ***commit*** token, and the change agreed upon is committed by each member as the token is circulated around the ring. The protocol ensures that each token is received by all members, processed only once, and never lost. More complete descriptions of the actions required by the different phases is covered in the following section.

a. Departure Processing

Once a member perceives the departure of its ***acwnbr***, voluntary or otherwise, a ***failagree*** token is initiated. The failed member is added to the status table with a ***failagree*** status. The token is incorporated into the token pool and transmitted via the FIFO channel to the ***cwnbr***. Similar processing occurs at members receiving a ***failagree*** token for the first time. Once the token has been received by the originator, the agree phase has been completed. The failed member is removed from the status table and group view. The token pool is purged. The ***failcomit*** token added to the token pool and transmitted around the ring. When a member receives a ***failcomit*** token for the first time, similar processing occurs.

b. Join Processing

The protocol maintains a logical marker between the first and last members to join the group. The first member is called the *host*. A new member will always join the group as the ***acwnbr*** of the *host*. The *host* has the responsibility of initiating the ***joinagree*** token for the new member. However, the *host* may not be the member of the group that receives a join request message from the new member. In this case, the message is converted to a token, forwarded, and the new member is added to the status table. Once the *host* has received either a ***joinreqst*** token or message, it initiates a ***joinagree*** token. The *host* then adds the token to the token pool, the member to the status table, and transmits the token to its ***cwnbr***. Similar processing occurs when a member receives a ***joinagree*** token for the first time. When the *host* receives the ***joinagree*** token via the token ring, it initiates the commit phase.

The ***joincomit*** phase consists of purging the token pool, incorporating the joincomit token in the token pool, adding the new member to the group view, and forwarding the token. Additionally, the host will transmit the status table, group view, and token pool to the new member in the initial parameters message, ***initparams***. This is accomplished by ***IntegrateMember***.

III. PROTOCOL CHANGES

This chapter describes the revisions to the group membership protocol proposed in [5]. Modifications discussed include lost messages, delayed transmission, lost token acknowledgments, and proper termination of the agree phase.

A. LOST INITIAL PARAMETERS MESSAGE

1. Problem

Consider a lost initial parameters ***initparam*** message. After the ***initparam*** message is sent, the joining member is regarded as part of the group by the sender even if it is lost. However, the group membership, token pool and status table are not accurately reflected in the joining member's local database. There was no mechanism for re-transmission of the ***initparam*** message, nor was it possible to recreate the information locally.

2. Solution

The status reporter is required authenticate the group membership. Reports are generated in response to status queries from only those members that are in the group or are joining the group. Status queries from outside the group are ignored. See Figure 2.

```
ReportStatus process at $p_i$
if (not blocked by IntegrateMember)
    if (querying member $\in GV_{p_i}$ or has joinagree status)
        $p_{mon}$ = querying member
        send status to $p_{mon}$
        if (previous querying member = $p_{mon}$)
            send TokenPool($p_i$) to $p_{mon}$
        end
    end
end
end ReportStatus
```

Figure 2 Reporting of Status

3. Justification

A lost ***initparam*** message will result in the new member failing to respond to the host's first ***statusqry***. Upon time out, the new member will be considered a failed process. The host's original ***acwnbr*** will then be monitored anew by the host. The new member, never having received the ***initparam*** message will time out on the join request. It will then attempt to join again.

By the time the ***initparam*** message is transmitted, the host's original ***acwnbr*** has knowledge that the new member is now joining the group. Therefore, with this change, the host's original ***acwnbr*** issues status reports to the new member's queries.

4. Side effects

Such group authentication will prevent multiple switching of the ***cwnbr***. If the ***initparam*** message is lost, the host's original ***acwnbr*** will be un-monitored for a short time. Upon the failure detection of the new member, the host's original ***acwnbr*** will again be monitored by the host process. To ensure that the ***StatusReporter*** responds only to members within the current group view, ***IntegrateMember*** must be atomic with respect to ***StatusReporter***. This will prevent race conditions when ***initparam*** message is received, followed by an almost simultaneous receipt of the first status query from the host process. Figure 3 shows the process dependencies, while Figure 4 depicts the specification for ***IntegrateMember***. The inter-process dependencies for the monitor processes are shown in Figure 5.

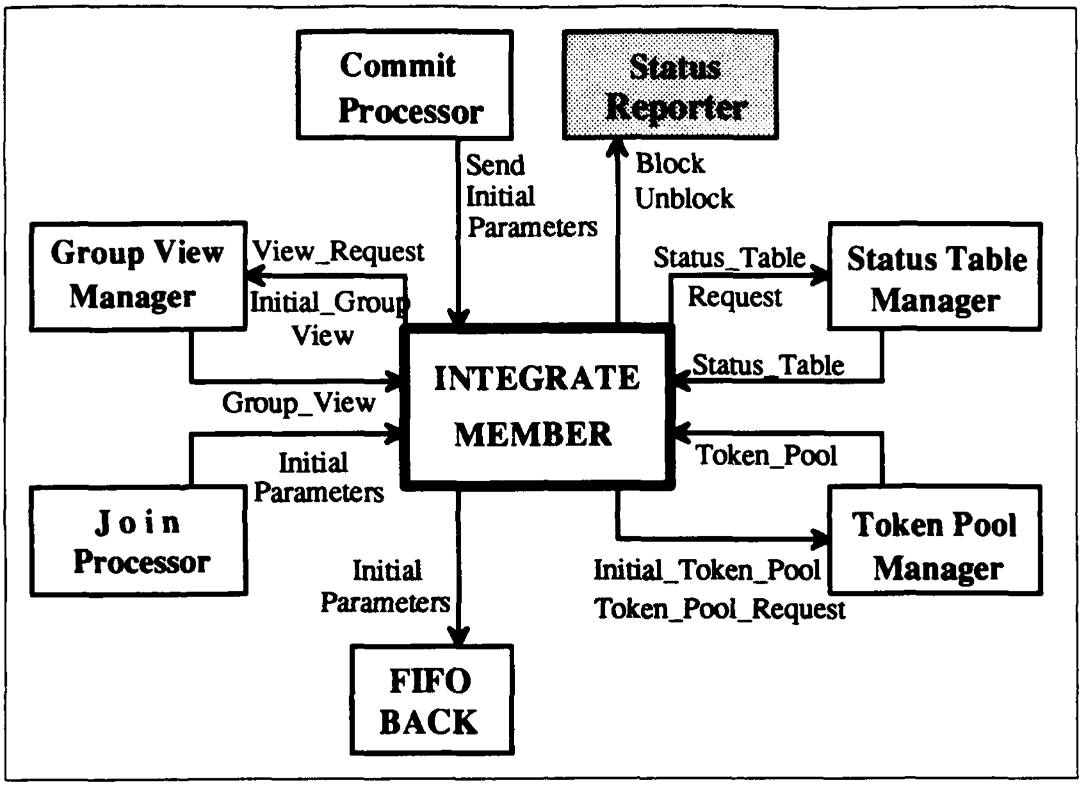

Figure 3 Integrate Member - Process Dependencies

```
IntegrateMember
if (initial parameters)
    send blocking message to status reporter
    send $GV_{P_i}$ to group view manager
    send unblocking message to status reporter
    send $ST_{P_i}$ to status table manager
    send $TokenPool(p_i)$ to token pool manager
else
    get $GV_{P_i}$ from group view manager
    get $ST_{P_i}$ from status table manger
    get $TokenPool(p_i)$ from token pool manager
    assemble initparam message
    send initparam message to new member
end
```

Figure 4 Integrate Member Process Specification

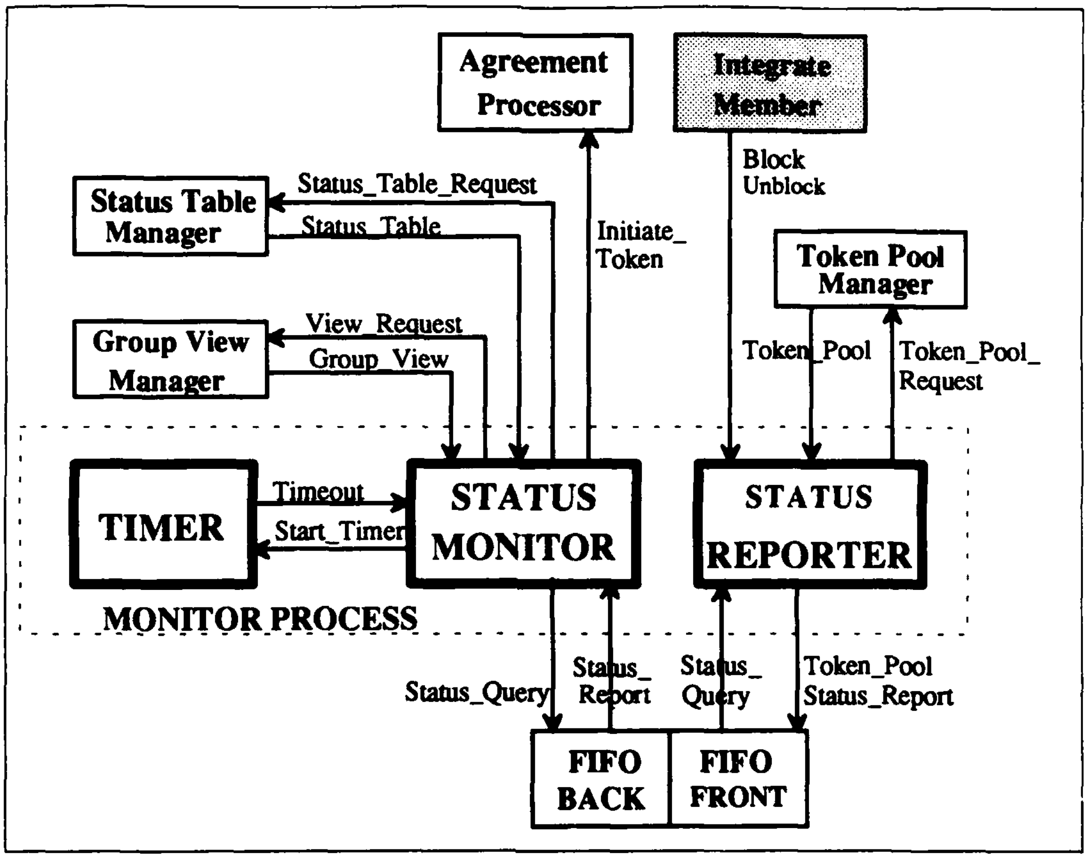

Figure 5 Monitor Process - Internal Structure and Dependencies

B. DUPLICATE PROCESSING

1. Problem

The protocol was designed for processing in a member to be concurrent. Consider ***AgreeProcessor*** and ***ComitProcessor***. It is possible for the ***AgreeProcessor*** to receive an ***agree*** token immediately followed by an external token pool containing the same token. The token may be converted to a ***commit*** token and forwarded to ***ComitProcessor***. Due to context switching, processing of the ***commit*** token may not be immediate. Further, ***AgreeProcessor*** begins processing the token pool which contains the copy of the original ***agree*** token. Since the processing of the commit token has not occurred, it is possible for the agree token from the token pool to be detected as requiring conversion. The subsequent processing of the duplicate commit is an error. Figure 6 depicts the problem.

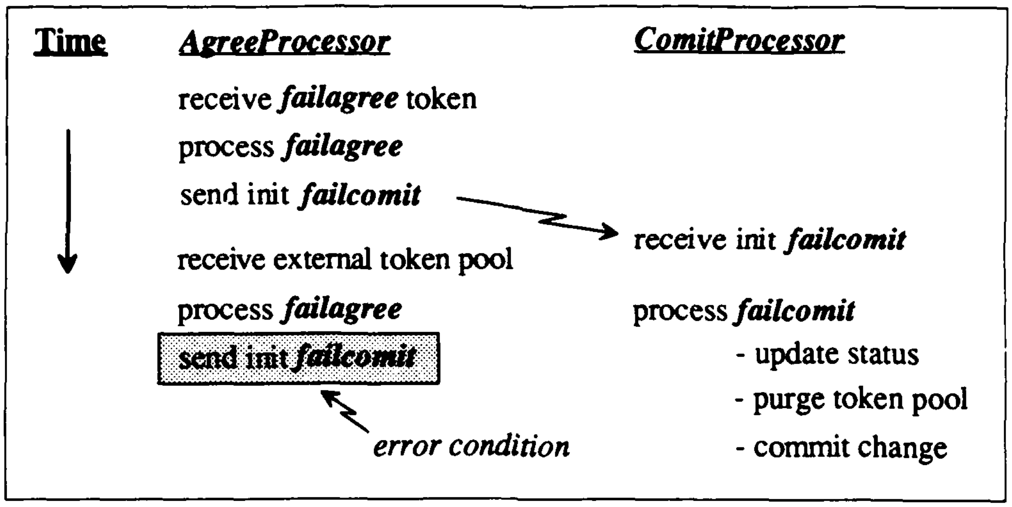

Figure 6 Error Condition Arising from Asynchronous Programs

2. Solution

In order to solve this problem, ***CommitProcessor*** must be atomic with respect to ***AgreeProcessor***. (see Figures 7 and 8)

3. Justification

Both processors use the same databases. Rejection of duplicate tokens depends upon the current state of the databases. Since token rejection is accomplished by ***AgreeProcessor***, and ***ComitProcessor*** updates the state to reflect the ***commit*** in progress, ***AgreeProcessor*** must not begin its next iteration until ***ComitProcessor*** has updated the state. ***ComitProcessor*** does not fully update the state until just prior to transmitting the ***commit*** token around the ring. Refer to Figure 9, lines 1-3.

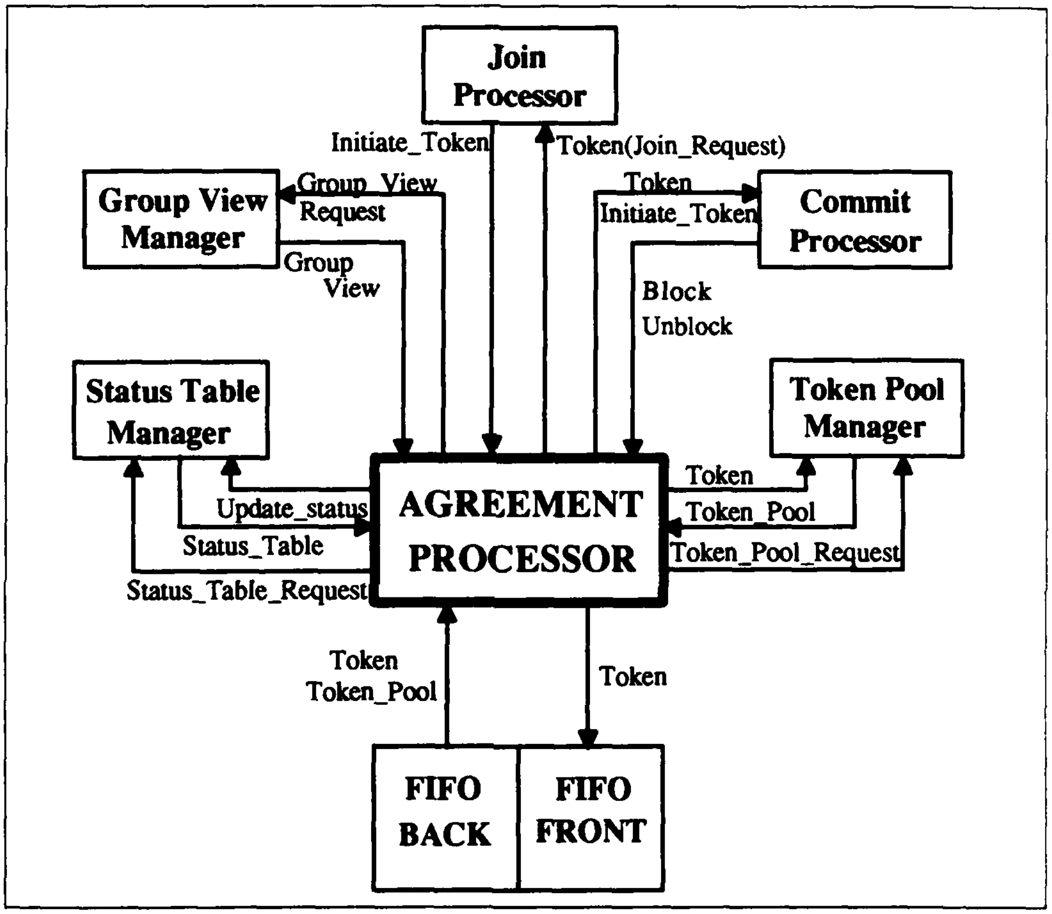

Figure 7 Agreement Processor - Process Dependencies

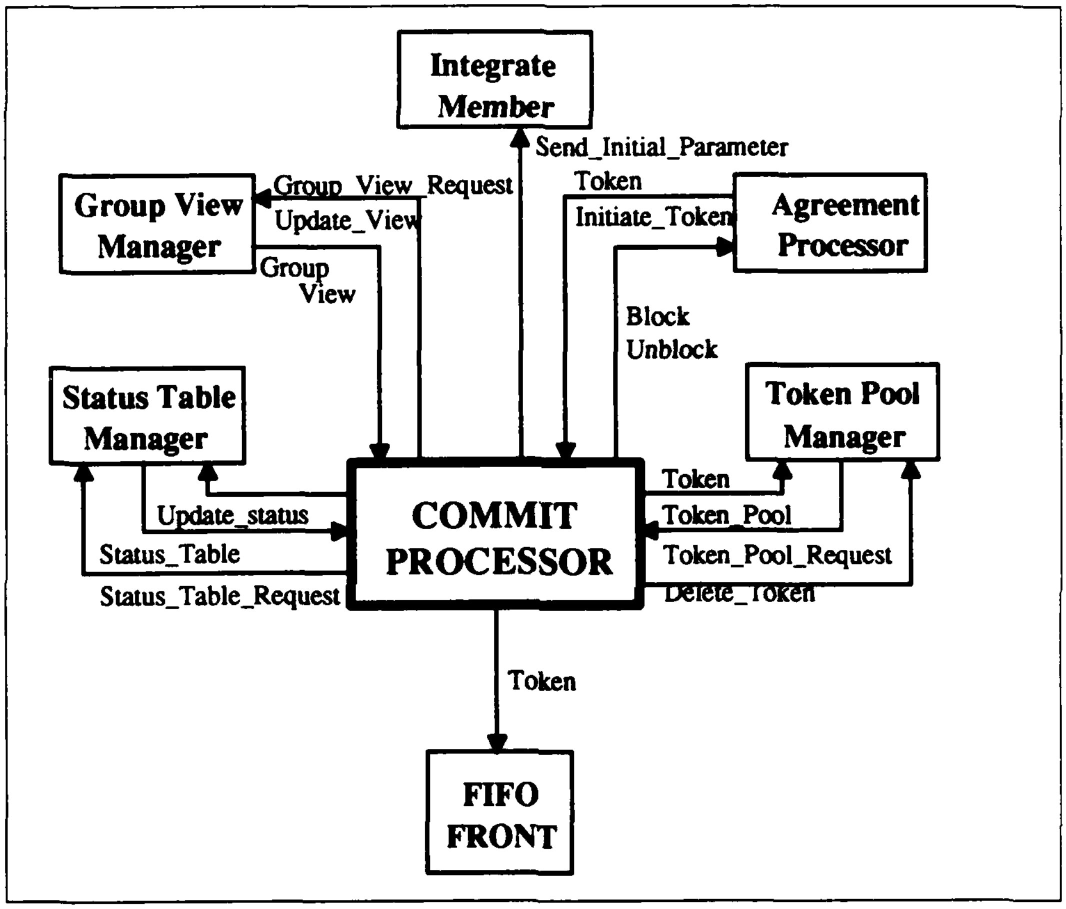

Figure 8 Commit Processor - Process Dependencies

```
   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 && $joinreq_{p_j}(p_k) \in TokenPool(p_i)$ )
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 $acwnbr(p_i)$
15 end
16 send $commit_{p_j}(p_k)$ token to $cwnbr(p_i)$

  end CommitChange
```

Figure 9 Actions for Committing a Change

C. INVALID DELETE TOKEN

1. Problem

An attempt to delete a nonexistent ***joinreqst*** token from the token pool would result in an error condition and would hang the process. The ***joinreqst*** token is not always present in the token pool. The token occurs only if a joining member has made the request to a member that is not the host.

2. Solution

Before attempting to delete the ***joinreqst*** token, the token pool is checked. If the token is present, it is then deleted. Figure 9, lines 5-7.

3. Justification

This is a special case token, and does not always occur in all token pools. Exception handling as above will correct any inconsistencies among the various token pools.

D. LOST TOKEN ACKNOWLEDGMENT

1. Problem

If the token acknowledgment is not received by the sending ***front*** process, the token will remain on the queue. It will be re-transmitted every time that ***front*** receives a message. The original token will be processed at the receiving end and further receipts of the same token will be detected as duplicates and discarded. The problem lies in that the queue is never cleared. Therefore, subsequent tokens will be blocked behind the token for which the acknowledgment was lost, unless a failure of one of the two members occurs.

2. Solution

This problem is solved by checking the serial number of the message on the receiving end of the FIFO channel. If the message is the last token received, the token acknowledgment is re-transmitted .

3. Justification

If the message received is the expected token, an acknowledgment is sent. The token is forwarded to the appropriate internal sub-process. If the last token received is received again, a token acknowledgment is sent back and the duplicate token is discarded. This mechanism will account and correct for lost token acknowledgments. Figure 10, lines 17-19.

```
FIFO Channel - BACK process

Wait for a channel ready to ready
if (internal channel ready)
    if (Status_Query)
        update acwnbr
        send Status_Query
    else if (Initial_Parameters)
        update acwnbr
        send Initial_Parameters
    else if (Join_Request)
        send Join_Request
    end
else /* external channel ready */
    if (message originator = acwnbr)
        if (Status_Report)
            send Status_Report to MONITOR_PROCESS
        else if (Token)
            if (Serial_Number = Expected_Serial_Number - 1)
                send Token_Ack  /* to acwnbr */
            end
            if (Serial_Number = Expected_Serial_Number )
                send Token to AgreeProcessor
                send Token_Ack  /* to acwnbr */
                increment Expected_Serial_Number
            end  /* out of order messages are discarded */
        else if (Token_Pool)  /* Token_Pool is always accepted */
            send Token_Pool to AgreeProcessor
            send Token_Ack  /* to acwnbr */
            set Expected_Serial_Number = Serial_Number + 1
        end
    end
end
```

Figure 10 FIFO Channel - Back Process

E. AGREEPROCESSOR

The specification for the ***AgreeProcessor*** was rewritten to account for various subtleties and to improve overall readability. All tokens received through the FIFO channel layer are sent to ***AgreeProcessor*** for dispatching to the appropriate processor. A duplicate token is one that has been previously processed at given member. Some scheme to detect and reject duplicate tokens is required. Proper termination of the ***agree*** phase and subsequent initiation of the ***commit*** phase are also required. Figure 11. In this section, we discuss the operation of the ***agree*** processor.

```
AgreeProcessor for $agree_{p_j}(p_k)$ at $p_i$
if (not blocked by CommitProcessor)
  if (initiate agreement message received) /* $p_i = p_j$ */
    add $agree_{p_j}(p_k)$ to $TokenPool(p_i)$
    $ST_{p_i}(p_k) \leftarrow$ joinagreed or failagreed
    send $agree_{p_j}(p_k)$ to cwnbr$(p_i)$
    send acknowledgment to calling process
  else /* a token or external token pool is received */
    if (ExtTokenPool)
      for $\forall$tokens $\in$ ExtTokenPool
        if (token $\in$ $TokenPool(p_i)$)
          if (originator failed)
            ProcessToken
          end
        else /* token not in TokenPool */
          if (received for the first time)
            ProcessToken
          end
        end
      end
    else /* a token was received */
      if (received for the first time)
        ProcessToken
      end
    end
  end
end
```

Figure 11 Agreement Processor

1. Initiate_Agreement Message

When ***AgreeProcessor*** receives an ***initiate_agreement*** message, the appropriate ***agree*** token is generated, added to the token pool and forwarded to the ***cwnbr***. The status table entry for the subject is updated. An acknowledgment is returned to the calling process. This reflects no change to the prior specification.

2. External Token Pool

When an external token pool is received, it is compared to the local token pool. All tokens in the external token pool are examined. Processing of a token depends on whether the token is also present in the local token pool.

If an ***agree*** token from the external token pool is in the local token pool, the token originator may have failed. Due to a failure of the originator, the ***agree*** token is requires conversion into a ***commit*** token at the first active clockwise neighbor of the originator only. Such tokens are processed as if they had been received as a separate token message. If the token is not present, it may have already been purged. These tokens must be rejected as duplicates.

a. Token Originator Failed

Detection of the token originator failing prior to initiating the ***commit*** phase is accomplished separately for ***joinagree*** and ***failagree*** tokens. Figure 12. It is necessary for the next active member to detect the originator's failure and also initiate the ***commit*** phase for the incomplete change started by the originator. The ***agree*** token may be received as part of the external token pool. The token will also be present in the local token pool from the ***acwnbr*** of the failed originator. This situation may also occur if a member in the middle of the ring fails. The failed member's ***cwnbr*** will receive an external token pool containing the original ***agree***. However, this does not require a ***commit*** to be initiated as the originator has not failed. It is essential that the differences be noted and accounted for. The same conditions are present, but different processing must occur.

The originator's failure during a ***joinagree*** phase can be detected if the rank of the external token pool originator is greater than the rank of the current member. Consider the host failing prior to initiating the ***commit*** phase. All members in the group have agreed to the ***join***. The ***joinagree*** token will be received by the new host upon ring reconfiguration. The new host's rank is zero (0) while the external token pool originator's rank is the (group size - 1). Since ***Rank***(originator) > ***Rank***(host), a ***commit*** must be initiated.

Consider member p_i with rank i failing during the ***joinagree*** phase. When the failure of p_i is detected, p_{i+1} receives the external token pool from p_{i-1}. The ***joinagree*** token is present in both the external token pool and *TokenPool*(p_{i+1}). The token is rejected since ***Rank***(p_{i-1}) < ***Rank***(p_{i+1}), .

```
LostAgreeToken
if (joinagree)
  if (rank($p_j$) > rank($p_i$))
    return true
  else
    return false
  end
end

if (failagree)
  if (RelativeRank($p_k$ , $p_i$) > RelativeRank($p_j$ , $p_i$))
    return true
  else
    return false
  end
end
```

Figure 12 Determination of Token Originator's Failure

Now we consider a duplicate ***failagree*** token. Define ***RelativeRank***(p_j, p_i) as the rank of p_j with respect to p_i instead of the host. A ring transversal starts and ends at the same specified process, i.e. any given member follows itself in a ring transversal. Figure 13. ***RelativeRank*** for a process that is not a member of the group is undefined. Recall the subject of a ***failagree*** remains a member of the group view until the ***commit*** is processed. A lost ***failagree*** token is determined by the ***RelativeRank*** of the token subject p_s and token pool originator p_{tpo}.

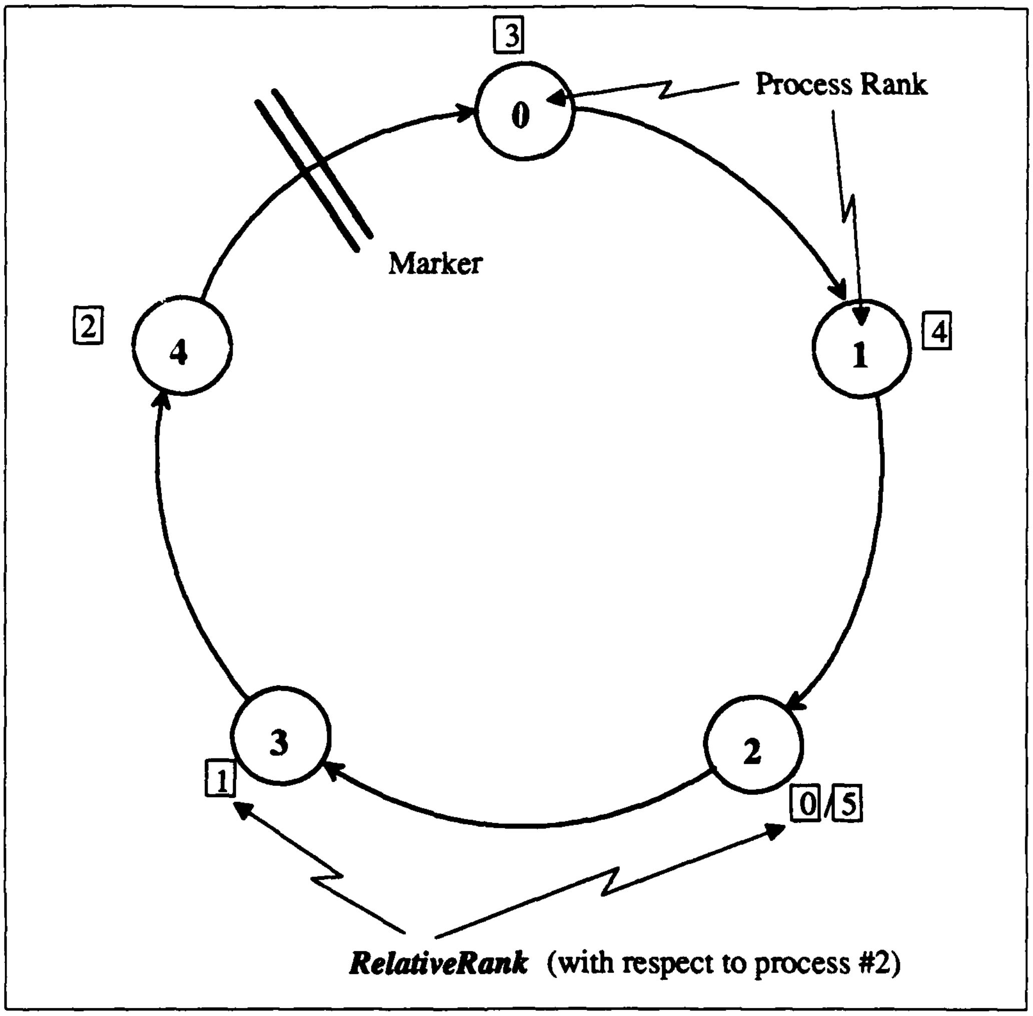

Figure 13 Relative Rank

If the ***RelativeRank***(p_s, p_i) ≥ ***RelativeRank***(p_{tpo}, p_i) the token originator has failed and the ***failcomit*** phase should be initiated. This situation will occur if the failed token originator itself fails prior to initiating the ***commit*** phase.

b. Duplicate Processing

Conditions to detect if a token has been received and processed already are summarized in Table 1. It is necessary to detect duplicate processing if the external token pool contains tokens not found in the local token pool. There are two possible ways for this to occur. If the token has not been received and processed, or the token has been purged from the local token pool.

Table 1 CONDITIONS TO DETECT DUPLICATE PROCESSING

Token	**Condition**
joinreqst ***joinagree*** ***joincomit***	$p_k \in GV_{p_i}$
failagree ***failcomit***	$p_k \notin GV_{p_i}$

Consider the ***commit*** phase. All members of the group have agreed to a particular ring reconfiguration. Due to the latency of token transmission around the ring, not all members commit the change simultaneously. Recall the token pool is purged of old ***commit*** tokens and the corresponding ***agree*** token prior to the ***commit*** token transmission. If a member that has committed a change receives an external token pool from a member that has agreed to the change, the ***agree*** token remains in the external token pool. Likewise, a previous ***commit*** token may be received as part of the external token pool. Since the tokens have been processed and removed from the local token pool, it is necessary to check the effects these tokens may have had on the group view, had they been previously processed. For duplicate ***join*** tokens, the subject would already be a part of the group view. Conversely, the subject of duplicate ***fail*** tokens would have already been removed from the group view. In this manner, duplicate processing can be detected and the tokens rejected.

It is not necessary to include all possible conditions for a token having been processed. Recall the duplicate processing check occurs only if the token is not present in the local token pool. For a ***joinreqst***, if the token is received as part of the external token pool and has been purged from the local token pool, the ***joincomit*** must have occurred. Since the result of the ***joincomit*** is subject becoming a part of the group, it is only necessary to check the end result. Intermediate stages of the ***join*** process have not deleted the ***joinreqs***t token from the local token pool. Since the token remains in the token pool, it is in both local and external token pools and is rejected. Similar logic results in the conditions presented in Table 1.

3. Tokens

Each type of token is handled individually upon receipt by ***AgreeProcessor***. Figure 14. ***AgreeProcessor*** acts as a filter to remove duplicate tokens before forwarding non-***agree*** tokens to the appropriate processor. ***Agree*** tokens are processed locally.

a. Joinreqst Tokens

The ***joinreqst*** tokens are forwarded to ***JoinProcessor*** for further processing.

b. Agree Tokens

If the current process is not the originator of the token, and it is not part of the local token pool, it is added to the token pool, the status updated and the token sent to the ***cwnbr***. This accounts for the first time an ***agree*** token is received and processed.

```
ProcessToken
if (joinreqst)
      send token to JoinProcessor
elseif (commit)
   send token to ComitProcessor
elseif (agree)
   if (($p_i \neq p_j$) && (agree token $\notin$ TokenPool($p_i$))
      add agree$_{p_j}$($p_k$) to TokenPool($p_i$)
      $ST_{p_i}(p_k) \leftarrow$ FailAgreed or JoinAgreed
      send agree$_{p_j}$($p_k$) to cwnbr($p_i$)
   else                    $p_j$
      if (($p_i = p_j$) || ($\forall p_l \mid p_l \rightarrow p_i, p_l \in ST_{p_i}$))
         compute rank $\forall p_l \in ST_{p_i}$ with Agreed status
         if rank($p_k$) = smallest
            send initiate_comit to ComitProcessor
         else
            $ST_{p_i}(p_k) \leftarrow$ joinpendg or failpendg
         end
      end
   end
end
```

Figure 14 Processing Agree Tokens

If the current process is the token originator, or the first active process clockwise from the originator that receives the ***agree*** token after it circulates around the ring, the rank of all processes with an ***agreed*** status is computed. An ***initiate_commit*** is transmitted if the subject is the lowest ranked of all processes fulfilling the above conditions. If not, the subject's status is updated to ***pending***.

c. *Commit Tokens*

A ***commit*** token is immediately sent to ***CommitProcessor*** for processing.

4. Side Effects

The external token pool mentioned above requires a new message type, *ExTknPool*. Figure 15. The message includes the originator of the token pool as an internal field. This is required by ***AgreeProcessor*** for determination of a failed token originator.

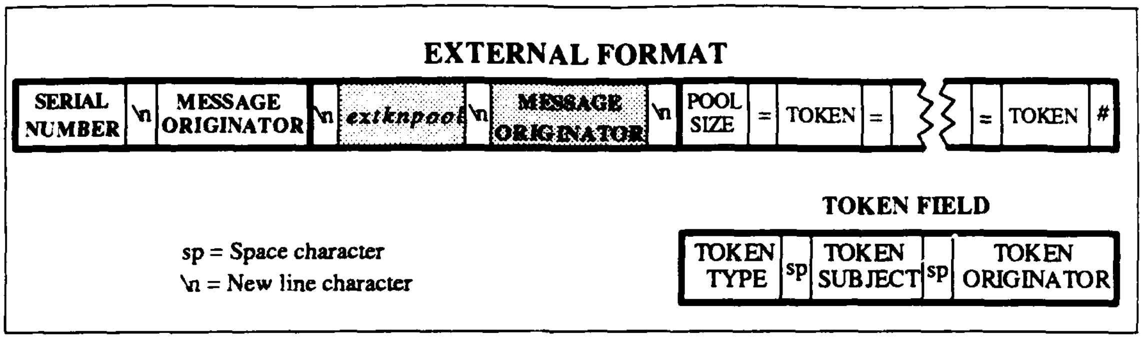

Figure 15 External Token Pool Message Format

F. MULTIPLE JOINS

1. Problem

Consider two members attempting to join a group almost simultaneously. Figures 16-18. The first member's ***join*** will be completed properly. Upon completion of the first ***join***, it is possible to begin processing the second member's ***joinreqst*** before the FIFO channels reflect the re-configured ring with the first member in it. It is possible to complete the second ***join*** before the channels are re-configured. This can happen because of the de-coupled protocol and FIFO channels. The host does not change its ***acwnbr*** until it initiates a status query to the last member. The FIFO channel determines a member's ***acwnbr*** as the target of the most recent ***statusqry***. The ***cwnbr*** is the originator of the most recent ***statusqry*** received. There is an inherent latency involved in the FIFO channel reconfiguration due to the timing considerations of subsequent ***statusqrys***. Thus, it is possible for a ***joinreqst*** from a second new member to complete both phases of the ***join*** process prior to FIFO channel reconfiguration. The first new member may never have processed the ***joinagree*** and ***joincomit*** for the second joining member prior to the second member being incorporated into the group view. When the first joining member determines its ***acwnbr*** and receives the external token pool, the ***joincomit*** token for the second member may be present. Processing the token will result in attempts to delete the corresponding nonexistent ***joinagree*** token never received by the first member and subsequent removal of the second member from the status table. Both of these are error conditions.

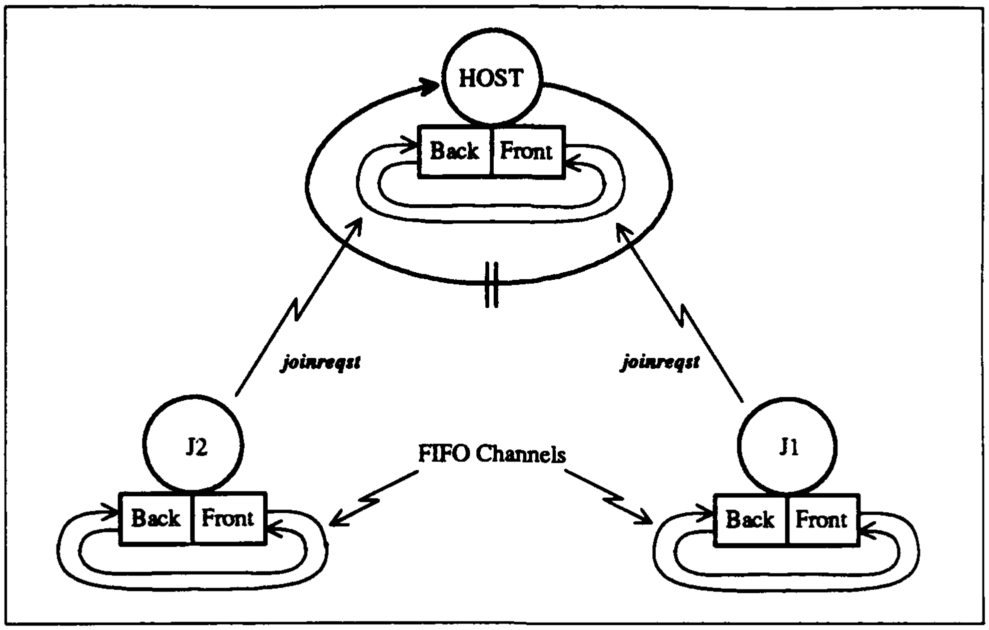

Figure 16 Simultaneous ***Joins***

2. Solution

Initiate the FIFO reconfiguration upon transmission of the initial parameters to the new member. Figure 10, line 7.

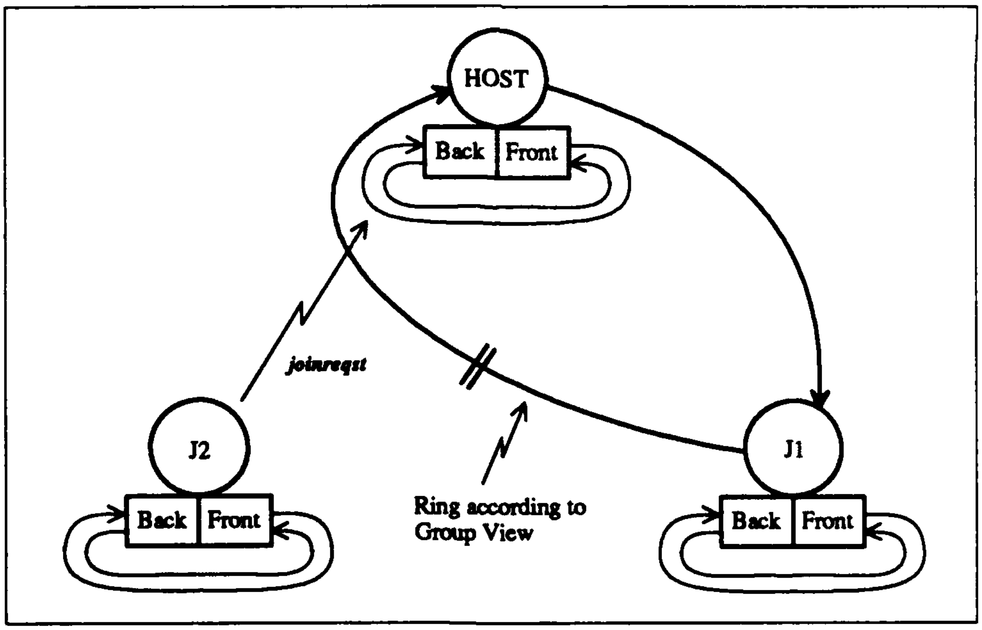

Figure 17 Group View after 1 member has joined

3. Justification

The FIFO channel can be considered to be re-configured when the host determines its ***acwnbr***. The response from the new ***acwnbr*** is not required, as the FIFO channel reject all tokens unless they are sent by the ***acwnbr***. Tokens from the old ***acwnbr*** are rejected and must be processed by the new ***acwnbr*** prior to being forwarded to the host member.

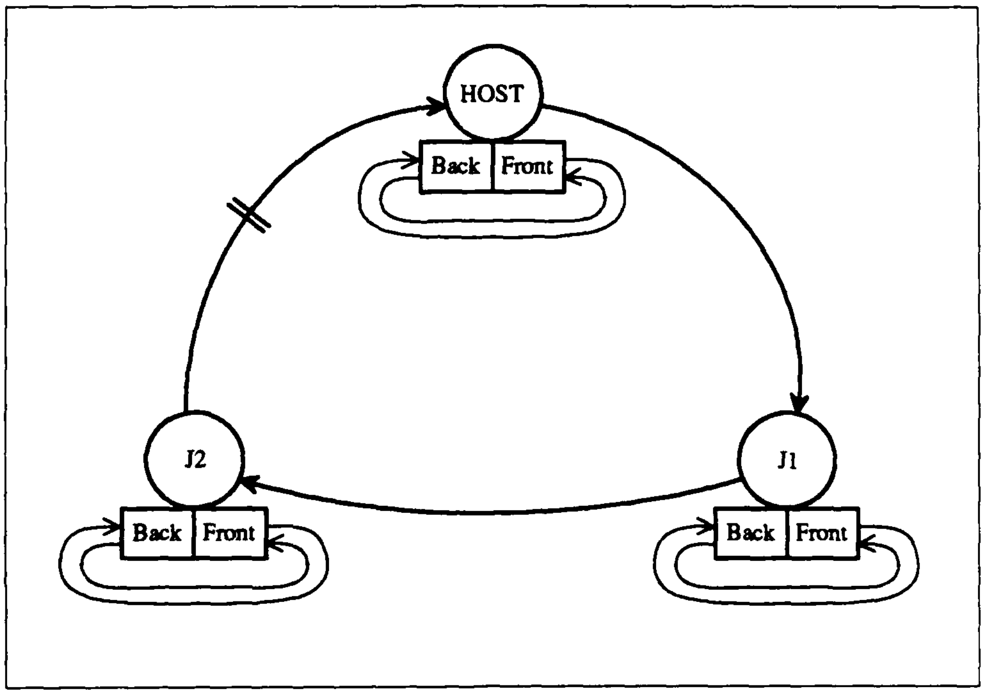

Figure 18 Group View at Host & J2

G. OTHER IMPROVEMENTS

In this section, oversights to the protocol specification and the modifications required are briefly described.

1. *Joinreqst* Token Processing

a. Problem

When processing a join message from a prospective member, ***InitiateJoin*** adds a ***joinreqst*** token to the token pool prior to generating it.

b. Solution

If the current message being processed is a join message, generate the ***joinreqst*** token and then add the token to the token pool. Figure 19, lines 10-13.

```
***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 AgreeProcessor for $p_{new}$
7        block until AgreeProcessor acknowledges end of processing
8     else
9        $ST_{P_i}(p_{new}) \leftarrow$ JoinRequested
10       if (join request message) /* $p_{new}$ locates $p_i$ and sends its join request */
11          generate $joinreq_{P_i}(P_{new})$ token
12       end
13       add $joinreq_{P_i}(p_{new})$ to $TokenPool(p_i)$
14       send joinreq token to $cwnbr(p_i)$
15    end
16 end
```

Figure 19 Processing of a Join Request Message / Token

2. *Commit* Token Generation

a. Problem

If a member was in a *pending* status, a commit token for that process was never generated prior to committing the change. These members would remain *pending* indefinitely.

b. Solution

Create the commit token before committing the change for a member with a *pending* status. Figure 20, lines 11-12.

```
ProcessCommitTkn for $commit_{p_j}(p_k)$ at $p_i$

if (initiate commit message received)
    generate commit token
    token to be processed ← generated token
else if (($p_i \neq p_j$) && (not duplicate))
    token to be processed ← received token
else
    exit
end
CommitChange
while ( $p_l \in ST_{p_i}$ with pending status & $Rank(p_l) < Rank(p_m)$, $p_m \in ST_{p_i}$)
    generate commit token
    token to be processed ← generated token
    CommitChange in rank order
end
```

Figure 20 Generate / Receive and Process a Commit Token

3. Message Queue in the FIFO Channel Layer

The front processor was modified to transmit the head of the message queue after receiving any message, either on the internal or external channel. Figure 21.

```
FIFO Channel - FRONT Process

Wait for a channel ready to read
if (external channel ready)
    if (Status_Query)
        send Status_Query to MonitorProcess
    else if (JoinRequest)
        send JoinRequest to JoinProcessor
    else if (InitialParameters)
        send InitialParameters to JoinProcessor
    else if (TokenAck)
        if (Received_Serial_Number = Expected_serial_number)
            remove Head_of_Queue
            decrement Queue_Counter
        end
    end
else /* internal channel ready */
    if (Token)
        change Token to external format  /* add external header */
        insert Token in queue
        increment Serial_Number
        increment Queue_Counter
    else if (TokenPool)
        discard all messages in queue
        change TokenPool to external format  /* add external header */
        insert TokenPool in queue
        increment Serial_Number
        increment Queue_Counter
    else if (StatusReport)
        update cwnbr
        send StatusReport to cwnbr
    end
end
if (Queue_Counter > 0)
    send Head_of_Queue to cwnbr
    set Expected_serial_number = Head_of_Queue_serial_number
end
```

Figure 21 FIFO Channel - Front Process

H. SYNOPSIS

This chapter has described the changes that were required to successfully implement the group membership protocol. Changes covered coding as well as protocol related problems not discovered in the original specification. These changes deal with the correct functioning of the protocol and do not address performance issues. Performance is dealt with in Chapter IV.

IV. PERFORMANCE OF THE GMP

The performance of the protocol on the Electrical and Computer Engineering Local Area Network (ECE LAN) consisting of SUN2 workstations connected via an Ethernet, was measured and the results are presented in this chapter.

A. LATENCY

The latency involved in processing changes to the group view is measured by each member using the local time clock on each specific processor. Timestamps were generated for the conditions listed in Table 2.

Table 2 CONDITIONS WARRANTING A TIME STAMP

Time Stamp	Where Taken in Code
t_{ai}	initiate an ***agree*** token
t_{ar}	receive an ***agree*** token
t_{cs}	send a ***commit*** token

The timestamps and related data were dumped to a file local to each processor. A filter program was designed and written to compile the data from the various systems, given a list of the members in the test group, and the maximum number of members. The filter program was designed for a restricted set of all possible group views given the above data. The case when the group view starts as the initial member, grows to the maximum number of members and then shrinks to the original host is the only possible case handled by the filter. Figure 22 illustrates the required group view changes.

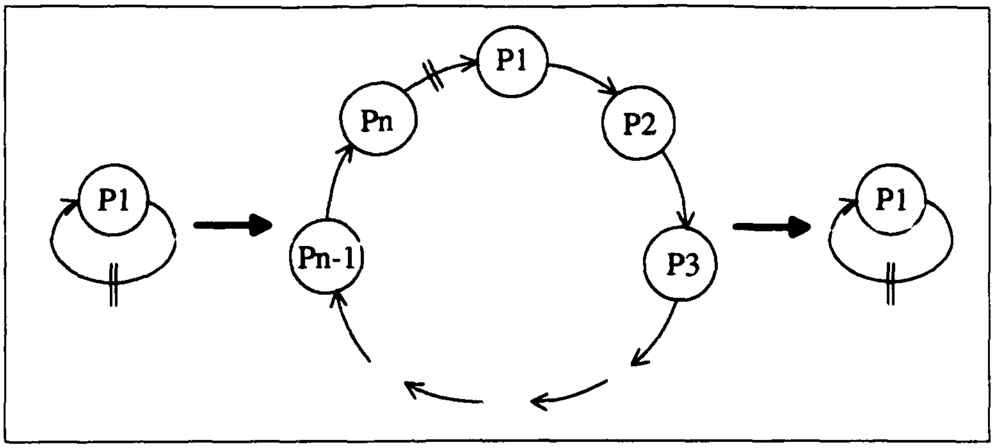

Figure 22 Group Changes Required by the Filter Program

A further restriction is that all members departing the group must be the host's ***acwnbr***. This removes the need for a local database in the filter program to track which members are part of the current iteratior of timestamp evaluations. Future improvements to the filter program can implement a dynamic database to account for all possible changes to the test group. The format of the output file is shown in Figure 23.

agree	sp	sec	sp	usec	**init**					
agree	sp	sec	sp	usec	**recv**	change to group size	sp	subject	sp	originator
comit	sp	sec	sp	usec	**send**					

sp = space

Figure 23 Time Stamp File Format

The time required to implement a change at a given member i is calculated by subtracting the initial time stamp from the completion time stamp.

$$t_i = t_{cs} - t_{ar} \mid i \neq host$$
$$t_i = t_{cs} - t_{ai} \mid i = host$$

The average time to commit a change at each member was calculated as follows:

$$\bar{t} = \frac{\sum_{i=1}^{n} t_i}{n}.$$

However, as the group increases from n members to $n+1$ members, the ***joinagree*** must be processed by the n members currently in the group. Similarly, the n members must receive and process the ***joincomit*** token. Hence, for a join to a group of *n* members, the processing time must be $n(t_{agree} + t_{comit})$, where t_{agree} is the time required to process an ***agree*** token and t_{comit} is the time to process a ***commit*** token. The communication time to transmit the token from one member to its neighbor is $t_{comm.}$ Since each of the tokens must be transmitted to the *n* members, the total communication cost is $t = 2n * (t_{comm})$. Therefore, the total time required to implement a ***join*** at all members of a *n* member group is

$$t = n(t_{agree} + t_{commit} + 2t_{comm})$$

Notice that the time to implement a change is proportional to the size of the group.

Table 4 PROCESSING TIME VALUES

Time	Occurrence
t_{agree}	time to process an agree token
t_{commit}	time to process a commit token
t_{comm}	inter-member communication time

Similarly, the expected time for a failure can be determined to have a linear relationship to the size of the remaining group.

B. TESTING

The performance of the GMP was tested on the ECE LAN consisting of SUN workstations linked via an Ethernet. There were no gateways between any of the members of the group. Only single complete changes were allowed at any given time. A complete reconfiguration included the underlying FIFO channel as well as the logical ring structure. A linear relationship was observed between the number of members in the group and the average time it took to commit the new member. Figure 24. Since the communication depends heavily upon the network load as well as the individual processor load, average values over a large variety of conditions such as time of day and number of people on the network were generated in order to get reliable data points.

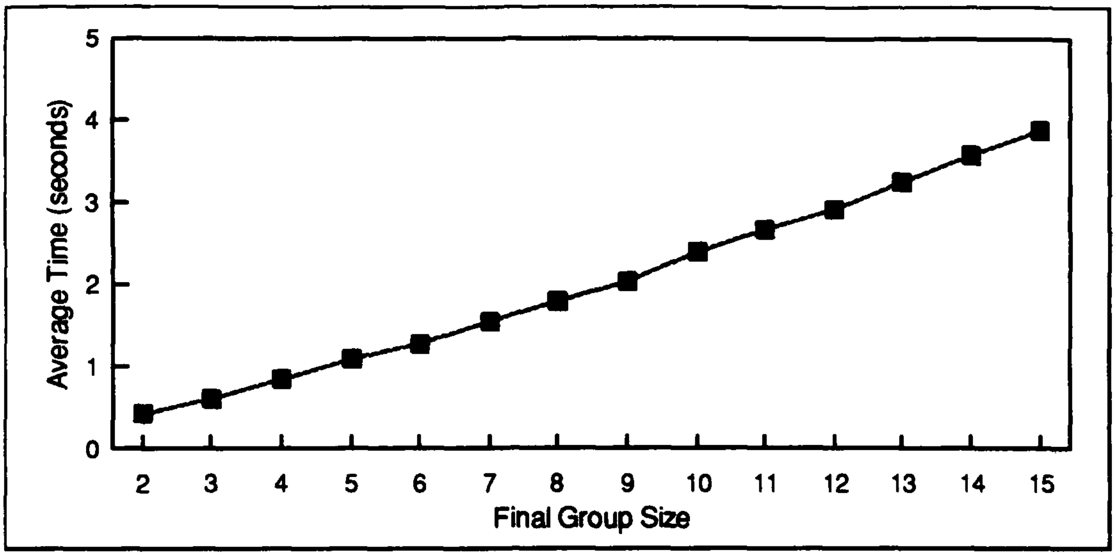

Figure 24 Average Time for each Member to Implement a Join to the Group

Similarly, the time required to remove a member from the group view was obtained and plotted. Thus, the relationship between time and group size was determined for a decreasing group size. Again, only single complete changes were allowed. Figure 25. A linear relationship was observed as expected.

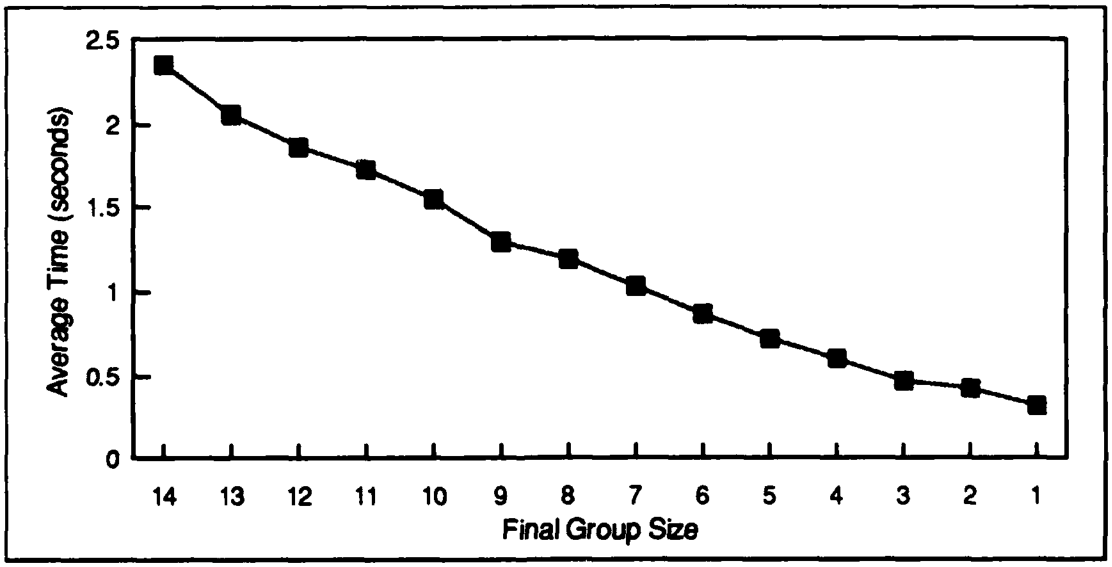

Figure 25 Average Time to Implement a Failure in the Group

V. METHODS FOR IMPROVING PERFORMANCE

In this chapter we explore two methods that might be used to further improve the overall performance of the protocol. Each could be incorporated along with the other or individually.

A. MESSAGE REDUCTION

The protocol as currently implemented has a high overhead due to the number of inter-member messages required to effect changes in the logical ring structure. In order to reduce the number of messages required for the maintenance of the logical ring, structure three methods are discussed below.

1. TokenPool versus Tokens

Consider multiple near simultaneous changes to the group view. Transmitting the token pool instead of individual tokens will result in a reduction of messages if the changes occur close enough together *such that the token pool for one change includes the* tokens for the subsequent changes. This will result in a decreased number of messages. However, the probability of such changes occurring is minimal. Since changes to the group are uncorrelated, the probability of such changes occurring is minimal. The corresponding reduction of message traffic is negligible. Additionally, there is an increase in the size of the message for most traffic. Modifications required to effect this include the dynamic generation of the group view by the FIFO channel. Instead of receiving the token pool for generation, the FIFO channel would receive a flag and, at that point, generate the external token pool message. One such method might be to set a flag that indicates that the token pool must be transmitted. FIFO properties are maintained by the order in which the tokens are processed upon receipt. Reduction will occur only if multiple changes occur prior to the transmission of the token pool for the initial change. Problems arise in the correct setting of the transmit flag; i.e., did change #2 get sent in the last token pool, or is another token pool transmit required. This method is not recommended since the number of messages is reduced only in special cases.

2. Periodic Token Pool

Recall that tokens are generated only if a change to the group view occurs. Another method that will reduce the number of inter-member messages is to periodically send the token pool instead of individual tokens. In this manner, multiple tokens can be transmitted simultaneously. Message reduction is indicated only if multiple changes to the group view occur within the period of the token pool transmission. If this does not occur, message traffic will actually increase; i.e., if there are no changes within this period the token pool is still transmitted. This method will also increase the latency in phase completion as tokens are not immediately forwarded around the ring. Additionally, the message size will be increased. This method is not recommended either.

3. Piggyback the Token Pool

A further refinement would be to include the local token pool as part of the status report. The monitoring member, upon receipt of a ***statusrpt***, would parse the token pool and process the appropriate tokens. FIFO channel requirements are maintained by the order in which tokens are processed upon receipt of a token pool. This, however, leads to additional processing for every status report.

Difficulties might occur in the latency of cycle completion in a large group. Most notably, consider when a new member has requested to join an existing group. Define t_l as the latency within a process. It is the difference in time between receiving the token pool via a ***statusrpt*** and transmitting the tokens around the ring. t_l can be modeled as a random variable. Ignoring the communication time, t_{comm}, and the processing time, t_{proc}, associated with the normal processing of tokens, the latency for a change to an N member group (i.e. a join request) becomes:

$$T_{total} = 2(N \times t_l)$$

Thus, the latency involved may become prohibitive for large N. This method is recommended for implementation, provided that the latency of changes is not important.

Care must be taken to ensure the time-out on a join request is large enough to encompass the worst case scenario. However, since the time-out is finite, this method place an implicit upper bound on the maximum group size. Once the group is large enough, new members will be unable to join due to the time needed to implement the ***join*** around the ring.

B. SINGLE-THREADED PROGRAM

1. Problem

The current design of the protocol involves concurrent processes handling specific areas of responsibility. Fully implementing this design would allow each process to reside on different processors. However, in most cases, a single processor is the norm. There is a significant amount of overhead due to the context switching and intra-member messages. The asynchronous nature of the protocol has several areas requiring process blocking as mentioned in chapter III.

2. Solution

Redesign the main process using a single-threaded program . Figure 26 shows the recommended processes and inter-dependencies.

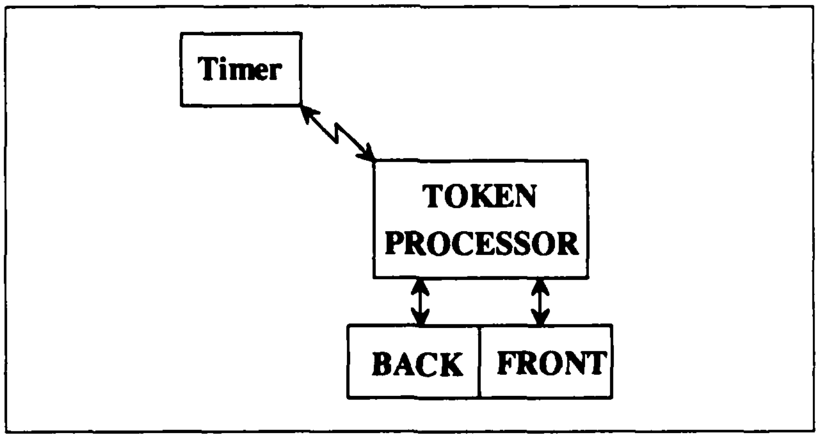

Figure 26 Single-Threaded Process Inter-Dependencies

3. Justification

The ***TokenProcessor*** would be a single-threaded program combining all aspects of the current design as shown in Table 4. The watchdog timer must still be a separate entity by definition. The FIFO channel is not included in the ***TokenProcessor*** as ***Back*** and ***Front***, are by nature, separate programs without any timing restrictions.

Table 4 SINGLE THREADED PROCESSES AND EQUIVALENTS

Single-Threaded GMP	**Multi-Threaded GMP**
Timer	***Timer***
	JoinProcessor
	AgreeProcessor
	ComitProcessor
	IntegrateMember
TokenProcessor	
	Status Table Manager
	Group View Manager
	Token Pool Manager
	StatusMonitor
	StatusReporter
BACK	***BACK***
FRONT	***FRONT***

The asynchronous nature of the design would be eliminated. The need for block and wait would be eliminated if a single-threaded program design were to be used. Additionally, the design would result in a significant decrease in the overhead costs due to the inter-process messages and connecting services being eliminated. The single-threaded program also eliminates the need for separate database managers. The TokenProcessor can maintain all databases internally, with different pointers keeping the different databases separate.

Concurrent with the new design, a review of all subroutines is recommended. Current design and programming practices preserves all data passed to the subroutines. This can lead to significant overhead since the data is stored multiple times.

VI. CONCLUSIONS AND RECOMMENDATIONS

In this thesis, the modifications required to implement the group membership protocol as proposed by [5] are presented. The protocol has been successfully implemented and to date has run continuously for more than 48 hours with a stable group membership. Additionally, a group size of 20 members was achieved. These results, though preliminary, are the first for this protocol. Although the protocol is functioning, continued debugging and improvement are currently going on.

As expected, the time required to implement a change was found to have a linear relationship to the eventual group size.

Further work should include the re-design of the protocol as a single-threaded program. In this manner, the response of the protocol can be improved as inter-process communication time is reduced drastically. However, attempting to implement the message reduction schemes to improve performance is not recommended as there is little to gain in the number of messages. On the contrary, implementation of the message reduction schemes would result in a large increase in the latency of changes to the membership.

Additional research is suggested in the area of network partitioning. Consider that a network may partition in two separate halves that are fully connected on either side of the boundary. A group originally existing on both sides will become two groups with the same name operating independently on either side of the partition. The difficulty arises when the network is repaired. The protocol does not provide for the possibility of merging the two groups back into the original group. The problem of handling network partitioning is non-trivial.

LIST OF REFERENCES

[1] 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 University.

[2] Kenneth P. Birman, "The process group approach to reliable distributed computing," Technical Report TR91-1216, Cornell University Computer Science Department, Ithaca, NY, July 1991.

[3] 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

[4] 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

[5] Fernando Pires, "Design of a Decentralized Asynchronous Group Membership Protocol and an Implementation of Its Communications Layer," Master's Thesis, March 1993, Naval Postgraduate School, Monterey, California

[6] 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.

[7] W. Richard Stevens, *Unix Network Programming*, Prentice Hall, 1990.

APPENDIX

The following code is included for completeness.

TABLE OF CONTENTS

DEFINITIONS

and

UTILITIES

```
/*
 * Definitions for GMP programs
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <netdb.h>
#include <sys/uio.h>
#include <errno.h>

#define CALLOC(n,type) (type *)calloc((unsigned) n,sizeof(type))
#define REALLOC(ptr,n,type) (type *)realloc(ptr,(unsigned) (n*sizeof(type)))

#define FRONT_PORT          5432
#define BACK_PORT           5433
#define SERV_HOST_ADDR      "131.120.20.102" /* host address for server sun2 */
#define UNIXSTR_PATH        "./s.unixstr"
#define UNIXSTR_TMPL        "/tmp/so.XXXXXX"
#define MAXLINE             255
#define TRUE                1
#define FALSE               0
#define MAXPATH             15
#define MAXPORT             6
#define MAXFD               6
#define MAXNUM              6

/* The following definitions do not account for the terminating NULL */
#define MAXHOSTNAME  50
#define MAXIPNAME    15
#define MAXLMTSIZE   MAXIPNAME + 2*MAXPORT + 2
#define   BUFSIZE    512
#define NODATAMSG    10
```

```
#define TIMERMSG       16
#define HEADERSIZE     11

#ifndef INADDR_NONE
#define INADDR_NONE    0xffffffff
#endif /* INADDR_NONE */

#define INVALDMSG      0
#define TOKENTOKN      1
#define TOKENPOOL      2
#define TOKENACKN      3
#define STATUSQRY      4
#define STATUSRPT      5
#define STATUSTBL      6
#define INITPARAM      7
#define JOINREQST      8
#define UPDSTATUS      9
#define GROUPVIEW      10
#define UPDATVIEW      11
#define SNDINIPAR      12
#define INITTOKEN      13
#define VIEWREQST      14
#define STATREQST      15
#define TOKPREQST      16
#define INITGVIEW      17
#define INITTABLE      18
 define INITPOOL       19
#define TIMEOUT__      20
#define STARTTIMR      21
#define DELTTOKEN      22
#define EXTKNPOOL      23
```

Table of Contents

```
/*********************************************************************
* QUEUE HANDLING AUXILIARY FUNCTIONS FOR FIFO PROCESSES
*        (FIFOUTIL.C)
*
*        The following functions are available to be used:
*        void enqueue(queue *quptr, char *msg);
*        void dequeue(queue *quptr);
*        void get_queue_head(queue *quptr, char **msg);
*        void flush_queue(queue *quptr);
*        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
*
**********************************************************************
*  Writen by:     Fernando J. Pires
*                 David Pezdirtz
*
*  Last revision:   12 Jul 1993
*
**********************************************************************/
```

```
/*********************************************************************
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(head, tail, msg)
link **head, **tail;
char *msg;

{
 link     *ptrlmnt, *tmp;
 ptrlmnt = CALLOC(1, link);
 ptrlmnt->data = CALLOC(strlen(msg) + 1, char);
 strcpy(ptrlmnt->data, msg);
 ptrlmnt->next = NULL;

 if (*head == NULL) {
   *head = ptrlmnt;
   *tail = ptrlmnt;
 }
 else {
   tmp = *tail;
   tmp->next = ptrlmnt;
   *tail = ptrlmnt;
 }

} /* end enqueue */
```

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

    Sample call:   dequeue(quptr);

****************************************************************/
void dequeue(head, tail)
link **head, **tail;
{
  link      *tmp;

  if (*head != NULL) {
    tmp = *head;
    *head = tmp->next;

    if (head == NULL){
            *tail = NULL; }

    free(tmp->data);
    free(tmp);
  } /* end if quptr */

} /* end dequeue */
```

```
/****************************************************************
  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(head, msg)
link *head;
char **msg;
{
  if (head == NULL){
    *msg = NULL; }
  else {
    *msg = head->data; }
} /* end get_queue_head */
```

```
/************************************************************
  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(head, tail)
link **head, **tail;
{
  link      *tmp;

  while (*head != NULL) {
    tmp = *head;
    *head = tmp->next;

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

  *tail = NULL;

} /* end flush_queue */
```

```
/************************************************************
  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_short port;

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

  list = str2list(tmp, ";");

  if ( getfromlist(list, &IPaddr, 1) != 1) {
    printf("sen_msg_front: IP address error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(list, &frontport, 2) != 2) {
    printf("sen_msg_front: front port error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* convert port # to network format */
  port = htons( (u_short)atoi(frontport) );

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

  removelist(list);
  free(tmp);
} /* end send_msg_front */
```

```
/***************************************************************
  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_short port;

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

  list = str2list(tmp, ";");

  if ( getfromlist(list, &IPaddr, 1) != 1) {
    printf("send_msg_back: IP address error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(list, &backport, 3) != 3) {
    printf("send_msg_back: back port error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* convert port # to network format */
  port = htons( (u_short)atoi(backport) );
  senmsg(msg, strlen(msg), IPaddr, port);

  removelist(list);
  free(tmp);
} /* end send_msg_back */
```

```
/***************************************************************
  send_ack - assembles an ack as a reponse to an external
  mes:age '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);
  strcpy(tmp,msg);

  /* assemble ack message */
  list = str2list(tmp, "\n");

  if ( getfromlist(list, &ackmsg, 1) != 1) {
    printf("send_ack: serial number in error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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);
} /* end send_ack */
```

```
/****************************************************************
  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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

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

} /* end send_msg_in */
```

Table of Contents

```
/***************************************************************************
 * Utilities for the protocol processes. (gmputil.c)
 *
 * Description: GetAcwnbr():
 *           InStatusTable():
 *           InGroup():
 *           CountDown():
 *           SendTkn2Agr():
 *           InTokenPool():
 *           TokensREqual();
 *           GetRank();
 *           GetTokenType();
 *           GetGroupSize();
 *           GetMembWithRank();
 *           GetStatus();
 *           RelativeRank();
 *           GetGroupView();
 *           GetStatusTable();
 *           GetTokenPool();
 *           agreetoken();
 *           comittoken();
 *           first_time();
 *
 * Written by:  Shridhar Shukla
 *              David Pezdirtz
 *
 * Date:     23 Nov 1993
 ***************************************************************************/

#define FAILAGREE       0
#define FAILCOMIT       1
#define JOINAGREE       2
#define JOINCOMIT       3
#define JOINRQSTT       4

#define NONBLOCKING     1
#define BLOCKING        0
#define GVLISTOFFSET    4    /* 4th field is the first element in gv list */
#define STLISTOFFSET    3    /* 3rd field is the first element in st list */
```

```
#define TKPLISTOFFSET    3    /* 3rd field is the first element in token pool list */
#define COUNT_DOWN_STEP   1000  /* 1 timer tick is AT LEAST 1000 micro seconds. */
#define RESENDREQST       10000  /* resend request after at least 10 seconds. */

#define TPAD_INTERVAL     250    /* .25 sec */
#define TQRY_INTERVAL     500    /* .5 sec */

/* The following definitions do not account for the terminating NULL */

#define MSGTYPELEN              9
#define TOKENTYPELEN            9
#define STSTYPELEN              9
#define QUERYLEN                MSGTYPELEN+2*MAXLMTSIZE+3
#define INITTOKENMSGLEN         2*MSGTYPELEN+2+MAXLMTSIZE+1
#define INITGVMSGLEN            MSGTYPELEN+2*MAXNUM+MAXLMTSIZE+4
#define TOKENLEN                2*MAXLMTSIZE+TOKENTYPELEN+3
#define TOKENMSGLEN             TOKENLEN+MSGTYPELEN+2
#define UPDTSTSMSGLEN           MSGTYPELEN+MAXLMTSIZE+STSTYPELEN+3
#define UPDTVIEWLEN             14+MAXLMTSIZE+1
#defir   DELTKNLEN              10+TOKENLEN+1
#defi    SNDINIPLEN             10+MAXLMTSIZE+1
```

```
#include "gmp.h"
#include "socutil.c"
#include "msgutil.c"
#include <sys/filio.h>
#include <sys/time.h>
#include <time.h>

int GetAcwnbr();
int InStatusTable();
int InTokenPool();
int GetMembWithRank();
int TokensREqual();
int InGroup();
int CountDown();
int GetRank();
int GetTokenType();
int GetGroupSize();
void SendTKn2Agr();
int GetStatus();
int RelativeRank();
char *GetGroupView();
char *GetStatusTable();
char *GetTokenPool();
int agreetoken();
int comittoken();
int first_time();
```

```
/*****************************************************************************
GetAcwnbr: returns 0 if acwnbr is correctly initialized to a member
           address as computed from the current group view, obtained from
           the group view manager at gvmsoc and the current status table
           obtained from stmsoc according to the rule in the paper. In all
           other cases, returns -1.

           The group view must contain at least one element - the member
           itself.  The first element is always the ring host. The status
           table must at least contain the number of elements, even if 0.
           It is assumed that myaddr is never in st. This function does not
           check for myaddr being part of st.
*****************************************************************************/
int GetAcwnbr(acwnbr, myaddr, stmsoc, gvmsoc)
char *acwnbr, *myaddr, *stmsoc, *gvmsoc;

{
 int      msglen, stmfd, gvmfd, gvsize, stsize, stmsglen, myrank,
          candidateposition, found, searchcomplete, myposition;
 char     *gvbuf, *stbuf, request[NODATAMSG + 1], *candidate, *nextmember,
          *gvsizestrng, *stsizestrng;
 link     *gv, *st;

 /*acquire group view */
 strcpy(request, "viewreqst#");
 msglen = strlen(request);
 gvmfd  = connectUN(gvmsoc);

 if ( writemsg(gvmfd, request, msglen) < msglen) {
   printf("\tGetAcwnbr: reqst to gvm failed.\n");
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

 if ( (msglen=readmsg(gvmfd,&gvbuf,"#")) < 0 ) {
   printf("\tGetAcwnbr: gv read failed.\n");
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

 gvbuf[msglen] = NULL;
```

```
close(gvmfd);

/*acquire status table */
strcpy(request, "statreqst#");
msglen = strlen(request);
stmfd  = connectUN(stmsoc);

if ( writemsg(stmfd, request, msglen) < msglen) {
  printf("\tGetAcwnbr: reqst to stm failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

if ( (msglen=readmsg(stmfd, &stbuf,"#")) < 0 ) {
  printf("\tGetAcwnbr: st read failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

stbuf[msglen] = NULL;

close(stmfd);

/*
 * make lists from the message buffers and get list sizes
 * for group view  and for status table
 */

st = str2list(stbuf, "\n=#");

if ( getfromlist(st, &stsizestrng, 2) == 0) {
  printf("\tGetAcwnbr: st msg parsing for size failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

stsize = atoi(stsizestrng);
removelist(st);
```

```
/*
 * Search for myaddr in gv and return the element preceding it
 * (modulo group size) that is NOT part of the status table.
 *
 */

myrank = GetRank(gvbuf, myaddr);

if ( myrank < 0) {
  printf("\tGetAcwnbr: parse on myaddr search failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

gv = str2list(gvbuf, "\n=#");

if ( getfromlist(gv, &gvsizestrng, 3) == 0) {
  printf("\tGetAcwnbr: gv msg parsing for size failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

gvsize = atoi(gvsizestrng);

if (myrank == 0)
  candidateposition = gvsize + GVLISTOFFSET - 1;

else
  candidateposition = myrank + GVLISTOFFSET - 1;

if ( getfromlist(gv, &candidate, candidateposition) == 0) {
  printf("\tGetAcwnbr: parse on nbr search failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

/*
 * Check st for every group member anticlockwise from  myaddr
 */

found = FALSE;  /* found the Acwnbr */
searchcomplete = FALSE; /* finished looking but did not find one */
```

```
if (stsize == 0) {
  found = TRUE;
  strcpy(acwnbr, candidate); }

while ( (found == FALSE) && (searchcomplete == FALSE) ) {
  free(stbuf);

  /*acquire status table */
  strcpy(request, "statreqst#");
  stmsglen = strlen(request);
  stmfd = connectUN(stmsoc);

  if ( writemsg(stmfd, request, stmsglen) < stmsglen) {
          printf("\tGetAcwnbr: reqst to stm failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

  if ( (stmsglen=readmsg(stmfd, &stbuf,"#")) < 0 ) {
          printf("\tGetAcwnbr: st read failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

  stbuf[stmsglen] = NULL;

  close(stmfd);

  if ( InStatusTable(stbuf, candidate, (char *) NULL) == FALSE) {
    /* candidate is acwnbr if it is not in st */
          found = TRUE;
          strcpy(acwnbr, candidate); }

  else { /* rotate ring position */

          if ( candidateposition == GVLISTOFFSET)
            candidateposition = gvsize + GVLISTOFFSET - 1;

          else
            candidateposition--;

      /* all others in st. */
          if ( candidateposition == (myrank + GVLISTOFFSET)) {
            searchcomplete = TRUE;
            strcpy(acwnbr, myaddr); }

          else
            if ( getfromlist(gv, &candidate, candidateposition) == 0) {
              printf("\tGetAcwnbr: parse on nbr search failed.\n");
              printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(-1); }

          } /* end else InStatusTable */

  } /* end while found */

  removelist(gv);
  free(gvbuf);
  free(stbuf);

  if (found || searchcomplete)
    return(0);

  else {
    printf("\tGetAcwnbr: expected flow of execution failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

} /* end GetAcwnbr */
```

```
/****************************************************************************
InStatusTable: returns TRUE if candidate is in the status table, FALSE if not.
   Initializes status to the value from the status table unless the address
   passed as status is NULL,

   When status is not required:  InStatusTable(stbuf, candidate, (char *) NULL)
   When status is required:      InStatusTable(stbuf, candidate, status)
                                 where status is the address of a memory
                                 allocated null terminated string of 10
                                 characters including the null character.

****************************************************************************/
int InStatusTable(stbuf, candidate, status)
char *stbuf, *candidate, *status;
{
  int      nextinst, stsize, found;
  char     *stmember, *stsizestrng, *sts, *sttble;
  link     *st;

  /* copy the status table */
  sttble = CALLOC(strlen(stbuf) + 1, char);
  strcpy(sttble, stbuf);

  /* separate elements from their status */
  st = str2list(sttble, "\n= #");

  if ( getfromlist(st, &stsizestrng, 2) == 0) {
    printf("\tInStatusTable: st msg parsing for size failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  stsize = atoi(stsizestrng);

  if (stsize == 0)
    found = FALSE;

  else { /*search the status table entries */
    found = TRUE;
    nextinst = STLISTOFFSET; /*first member in st is the third field */

    if ( getfromlist(st, &stmember, nextinst) == 0) {
          printf("\tInStatusTable: first parse on st failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    while ( (found == TRUE) && (strcmp(stmember, candidate) != 0 ) ) {
          nextinst += 2; /* skip the status field */

          if ( getfromlist(st, &stmember, nextinst) == 0)
            found = FALSE;

    } /* end while found */

    if (status != NULL) {

          if (found == TRUE) {

            if (getfromlist(st, &sts, nextinst + 1) == 0) {
              printf("\tInStatusTable: initializing status failed.\n");
              printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(-1); }

            strcpy(status, sts);

          } /* end if found */

    } /* end if status */

  } /* end else stsize */

  removelist(st);
  free(sttble);

  return(found);

} /* end InStatusTable */
```

```
/**********************************************************************
CountDown:  Decrements the location at valptr after at least
            `step' microseconds.
**********************************************************************/
int CountDown(valptr, step)
int *valptr, step;
{
  usleep(step);
  return(*valptr = *valptr - 1);
} /* end CountDown */
```

```
/**********************************************************************
SendTkn2Agr: A token is sent to the agreement process and the caller blocks
      until the agreement process acks.
            exits upon failure to send.
**********************************************************************/
void SendTkn2Agr(agrsoc, tkntype, tknsubject)
char *agrsoc, *tkntype, *tknsubject;
{
  int     agrfd, msglen;
  char    c[1], initagree[INITTOKENMSGLEN + 1];

  /* assemble initiate agreement msg */
  strcpy(initagree, "inittoken\n");   /* header */
  strcat(initagree, tkntype);         /* type */
  strcat(initagree, " ");             /* separator */
  strcat(initagree, tknsubject);      /* subject fo token */
  strcat(initagree, "#");             /* end of msg */

  /* send initiate agreement msg and block */
  msglen = strlen(initagree);
  agrfd = connectUN(agrsoc);

  if ( writemsg(agrfd, initagree, msglen) < msglen) {
    printf("\tSendTkn2Agr: init agree send failed for %s.\n", tkntype);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( read(agrfd, c, 1) < 0) {
    printf("\tSendTkn2Agr: empty msg read failed after sending %s.\n", tkntype);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$.$$$$$$\n");
    exit(-1); }

  close(agrfd);

} /* end SendTkn2Agr */
```

```
/***************************************************************************
InGroup: returns the rank of the candidate from the group view. -1 if
      the candidate is not found.
******* *****************************************************************/
int InGroup(gvmsoc, candidate)
char *gvmsoc, *candidate;
{
  int       found, gvmfd, msglen, rank, gvsize;
  char      *gvbuf, *element, request[NODATAMSG + 1], *gvsiz.:strng;
  link      *gv;

  /*acquire group view */
  strcpy(request, "viewreqst#");
  msglen = strlen(request);
  gvmfd = connectUN(gvmsoc);

  if ( writemsg(gvmfd, request, msglen) < msglen) {
    printf("\tInGroup: reqst to gvm failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( (msglen=readmsg(gvmfd, &gvbuf,"#")) < 0 ) {
    printf("\tInGroup: gv read failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  gvbuf[msglen] = NULL;

  close(gvmfd);

  /*
   * make a list from the message buffer and get the host
   */
  gv = str2list(gvbuf, "\n=#");

  if ( getfromlist(gv, &gvsizestrng, 3) == 0) {
    printf("\tInGroup: msg parsing for gv size failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
  gvsize = atoi(gvsizestrng);
  rank  = 0;
  found  = FALSE;

  while ( (found == FALSE) && (rank <= gvsize-1) ) {

    if ( getfromlist(gv, &element, rank + 4) == 0) {
          printf("\tInGroup: gv parsing for element rank %d failed.\n", rank);
          printf("\07$$$$$$$$$$`$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    if ( strcmp 'element, candidate) == 0)
          fou.d = TRUE;

     else rank++;

   } /* end while found */

  removelist(gv);
  free(gvbuf);

  if ( found == TRUE )
    return(rank);

  else return(-1);

} /* end InGroup */
```

```
/***************************************************************************
GetMembWithRank: returns 0 and member is initialized to the element with rank.
      -1 if no member with the given rank exists.
            `member' is the address of the first character of the member string
            to be returned. Storage is allocated for `member' which should be
            freed by the caller.
            Incoming group view, gvbuf, must be a string.

            Sample call: char *member;
                      ...

           if ( GetMembWithRank(gvbuf, &member, n) != 0 )
                error processing ....;
           else {
                ...
           }
                ...

           free(member);
***************************************************************************/
int GetMembWithRank(gvbuf, member, rank)
char *gvbuf, **member;
int rank;

{
  int      gvsize;
  char     *gvsizestrng, *desiredmember, *gv;
  link     *gvlist;

  if (rank < 0) {
    printf("\tGetMembWithRank: Illegal value of rank %d . \n", rank);
    return(-1); }

  /* preserve original copy of gv */
  gv = CALLOC(strlen(gvbuf) + 1, char);
  strcpy(gv, gvbuf);

  /* Convert the gv string to a list */
  gvlist = str2list(gv, "\n=#");
```

```
  if ( getfromlist(gvlist, &gvsizestrng, GVLISTOFFSET - 1) == 0) {
    printf("\tGetMembWithRank: gv parsing for gv size failed.\n");
    printf("\tGetMembWithRank: gv = |%s\n", gvbuf);
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

  gvsize = atoi(gvsizestrng);

  if (rank > gvsize - 1) {
   printf("\tGetMembWithRank: rank %d exceeds gvsize - 1 %d . \n", rank, gvsize - 1);
   removelist(gvlist);
   free(gv);
   return(-1); }

  else {
   *member = CALLOC(MAXLMTSIZE + 1, char);

    if ( getfromlist(gvlist, &desiredmember, rank + GVLISTOFFSET) == 0) {
      printf("\tGetMembWithRank: gv parsing for member of desired rank failed.\n");
     printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

    strcpy(*member, desiredmember);
    removelist(gvlist);
    free(gv);
    return(0);

  } /* end else rank */

} /* end GetMembWithRank */
```

```
/*****************************************************************************
InTokenPool: returns TRUE if token is in the token pool, FALSE if not.
        For token equality, only checks the subject and the type,
         and ignores the originator.
*****************************************************************************/
int InTokenPool(tokenpool, token)
char *tokenpool, *token;

{
  int     found, next, done;
  char    *candidate, *poolsizestrng, *tp;
  link    *tkplist;

  /*
   * make a list from the message buffer and get the pool size
   */
  tp = CALLOC(strlen(tokenpool) + 1, char);
  strcpy(tp, tokenpool);

  tkplist = str2list(tp, "\n=#");

  found = FALSE;
  done = FALSE;

  next = TKPLISTOFFSET; /* first member in token pool is the third field */

  while ( done == FALSE){

     if ( getfromlist(tkplist, &candidate, next) == 0)
       done = TRUE;
     else{
       if (TokensREqual(token, candidate) == TRUE ) {
         done = TRUE;
         found = TRUE;
       }
     }
     next++;

  } /* end while !done */

  removelist(tkplist);
  free(tp);

  return(found);

} /* end InTokenPool */
```

```
/****************************************************************************
TokensREqual: returns TRUE if token has the same subject and type as
        candidate, FALSE if not.
****************************************************************************/
int TokensREqual(token, candidate)
char *token, *candidate;

{
  int       equal;
  char      *token1, *token2, *t1_type, *t2_type, *t1_subj, *t2_subj;
  link      *list1, *list2;

  /*
   * make local copies first to create lists
   */
  token1 = CALLOC(strlen(token) + 1, char);
  strcpy(token1, token);

  token2 = CALLOC(strlen(candidate) + 1, char);
  strcpy(token2, candidate);

  list1 = str2list(token1, " ");
  list2 = str2list(token2, " ");

  if ( getfromlist(list1, &t1_type, 1) == 0) {
    printf("TokensREqual: token type parsing for token 1 failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(list1, &t1_subj, 2) == 0) {
    printf("TokensREqual: token subject parsing for token 1 failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(list2, &t2_type, 1) == 0) {
    printf("TokensREqual: token type parsing for token 2 failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(list2, &t2_subj, 2) == 0) {
    printf("TokensREqual: token subject parsing for token 2 failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ((strcmp(t1_type, t2_type) == 0 ) && (strcmp(t1_subj, t2_subj) == 0))
    equal = TRUE;
  else
    equal = FALSE;

  removelist(list1);
  removelist(list2);

  free(token1);
  free(token2);

  return(equal);

} /* end TokensREqual */
```

```
/*****************************************************************************
GetRank: returns the rank of the candidate from the group view. -1 if
      the candidate is not found. Requires the caller to supply a
      string containing the group view.
*****************************************************************************/
int GetRank(gv, candidate)
char *gv, *candidate;

{
  int      found, gvlen, rank, gvsize;
  char     *lgv, *element, *gvsizestrng;
  link     *gvl;

  gvlen = strlen(gv);
  lgv = CALLOC(gvlen+1, char);
  strcpy(lgv, gv);

  /*
   * make a list from the local copy and search for the candidate
   */

  gvl = str2list(lgv, "\n=#");

  if ( getfromlist(gvl, &gvsizestrng, 3) == 0) {
    printf("\tGetRank: msg parsing for gvsize failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  gvsize = atoi(gvsizestrng);
  rank   = 0;
  found  = FALSE;

  while ( (found == FALSE) && (rank <= gvsize - 1) ) {

    if ( getfromlist(gvl, &element, rank + 4) == 0) {
          printf("\tGetRank: gvl parsing for element rank %d failed.\n", rank);
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    if ( strcmp(element, candidate) == 0)
          found = TRUE;
    else
          rank++;

  } /* end while found */

  removelist(gvl);
  free(lgv);

  if ( found == TRUE )
    return(rank);
  else
    return(-1);

} /* end GetRank */
```

```
/**************************************************************************
GetTokenType: does not disturb the token passed in.
        returns -1 if the token type is invalid.
**************************************************************************/
int GetTokenType(token)
char *token;
{
  int     type;
  char    ltkn[TOKENLEN + 1], *tkntype;
  link    *tknl;

  strcpy(ltkn, token);
  tknl = str2list(ltkn, " ");

  if ( getfromlist(tknl, &tkntype, 1) == 0 ) {
    printf("\tGetTokenType:  local token parsing for type failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1);}

  type = -1;

  if (strcmp(tkntype, "failagree") == 0)
    type = FAILAGREE;
  if (strcmp(tkntype, "failcomit") == 0)
    type = FAILCOMIT;
  if (strcmp(tkntype, "joinagree") == 0)
    type = JOINAGREE;
  if (strcmp(tkntype, "joincomit") == 0)
    type = JOINCOMIT;
  if (strcmp(tkntype, "joinreqst") == 0)
    type = JOINRQSTT;

  removelist(tknl);

  return(type);

} /* end GetTokenType */
```

```
/*************************************************************************
GetGroupSize: returns the group view size.  Requires the caller to supply a
        string containing the group view.
*************************************************************************/
int GetGroupSize(gv)
char *gv;

{
  int     gvsize, gvlen;
  char    *lgv, *gvsizestrng;
  link    *gvl;

  gvlen = strlen(gv);
  lgv = CALLOC(gvlen+1, char);
  strcpy(lgv, gv);

  /*
   * make a list from the local copy
   */
  gvl = str2list(lgv, "\n=#");

  if ( getfromlist(gvl, &gvsizestrng, 3) == 0) {
    printf("\tGetGroupSize: msg parsing for gvsize failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  gvsize = atoi(gvsizestrng);
  removelist(gvl);
  free(lgv);

  return(gvsize);

} /* end GetGroupSize */
```

```
/**************************************************************************
GetStatus: returns the status of the candidate from the status table. -1 if
    the candidate is not found. Requires the caller to supply a string
    containing the status table.
**************************************************************************/
int GetStatus(st, candidate)
char *st, *candidate;
{
  int stat;
  char *status;

  stat = -1;

  status = CALLOC( 9 + 1, char);

  /* if subject is in status table */
  if (InStatusTable(st, candidate, status) == TRUE) {

    if ( strcmp(status,"failagree") == 0)
      stat = FAILAGREE;
    if ( strcmp(status,"failcomit") == 0)
      stat = FAILCOMIT;
    if ( strcmp(status,"joinagree") == 0)
      stat = JOINAGREE;
    if ( strcmp(status,"joincomit") == 0)
      stat = JOINCOMIT;
    if ( strcmp(status,"joinrqstd") == 0)
      stat = JOINRQSTT;
  }

  free(status);
  return(stat);
} /* end GetStatus */
```

```
/**************************************************************************
RelativeRank: returns the relative rank of the subject wrt origin.  Modulo
          gv_size.  The relative rank of the original process is zero.  Returns
          (-1) if either subject, or origin are not in group view.
**************************************************************************/
int RelativeRank(gv, subject, origin)
char *gv, *subject, *origin;
{
  int rank_s, rank_o, rr;

  rank_s = GetRank(gv, subject);
  rank_o = GetRank(gv, origin);

  /* calculate the rr (assuming subject and origin in gv) */
  rr = rank_s - rank_o;

  if (rr < 0){
    rr = rr + GetGroupSize(gv);
  }

  /* check for error condition */
  if ((rank_s < 0) || (rank_o < 0))
    return(-1);
  else
    return (rr);

} /* end RelativeRank */
```

```
/*****************************************************************************
GetGroupView: returns a pointer to the group view.  Requires gvmsoc as a
            parameter.

            Typical call:

                        buf = GetGroupView(gvmsoc);

                                    ...

                        free(buf);

*****************************************************************************/
char *GetGroupView(gvmsoc)
char *gvmsoc;
{
  int  msglen, gvmfd;
  char gvreq[NODATAMSG + 1], *gv;

  /* get local group view */
  strcpy(gvreq, "viewreqst#");
  msglen = strlen(gvreq);
  gvmfd  = connectUN(gvmsoc);

  if ( writemsg(gvmfd, gvreq, msglen) < msglen ) {
    printf("\tGetGroupView: group view request failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( (msglen=readmsg(gvmfd,&gv,"#")) < 0 ) {
    printf("\tGetGroupView: group view read failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  gv[msglen] = NULL;
  close(gvmfd);

  return(gv);
}
```

```
/*****************************************************************************
GetStatusTable: returns a pointer to the status table.  Requires stmsoc as a
            parameter.

            Typical call:

                        buf = GetStatusTable(stmsoc);

                                    ...

                        free(buf);

*****************************************************************************/
char *GetStatusTable(stmsoc)
char *stmsoc;
{
  int  msglen, stmfd;
  char streq[NODATAMSG + 1], *st;

  /* get status table */
  strcpy(streq, "statreqst#");
  msglen = strlen(streq);
  stmfd  = connectUN(stmsoc);

  if ( writemsg(stmfd, streq, msglen) < msglen ) {
    printf("\tGetStatusTable: status table request failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( (msglen=readmsg(stmfd,&st,"#")) < 0 ) {
    printf("\tGetStatusTable: status table read failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  st[msglen] = NULL;
  close(stmfd);

  return(st);
}
```

```
****************************************************************************
GetTokenPool: returns a pointer to the token pool.  Requires tpmsoc as a
          parameter.

          Typical call:

                    buf = GetTokenPool(tpmsoc);

                              ...

                    free(buf);

****************************************************************************/
char *GetTokenPool(tpmsoc)
char *tpmsoc;
{
  int  msglen, tpmfd;
  char tpreq[NODATAMSG + 1], *tp;

  /* get local token pool */
  strcpy(tpreq, "tokpreqst#");
  msglen = strlen(tpreq);
  tpmfd  = connectUN(tpmsoc);

  if ( writemsg(tpmfd, tpreq, msglen) < msglen ) {
    printf("\tGetTokenPool: token pool request failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( (msglen=readmsg(tpmfd,&tp,"#")) < 0 ) {
    printf("\tGetTokenPool: token pool read failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  tp[msglen] = NULL;
  close(tpmfd);

  return(tp);
}
```

```
/****************************************************************************
agreetoken: return TRUE if tokentype = FAILAGREE or JOINAGREE
****************************************************************************/
int agreetoken(token)
char *token;
{
  int tokentype;

  tokentype = GetTokenType(token);

  if ((tokentype == FAILAGREE) || (tokentype == JOINAGREE))
    return(TRUE);
  else
    return(FALSE);

} /* end agreetoken */



/****************************************************************************
committoken: return TRUE if tokentype = FAILAGREE or JOINAGREE
****************************************************************************/
int comittoken(token)
char *token;
{
  int tokentype;

  tokentype = GetTokenType(token);

  if ((tokentype == FAILCOMIT) || (tokentype == JOINCOMIT))
    return(TRUE);
  else
    return(FALSE);

} /* end comittoken */
```

```
/***************************************************************************
first_time:

return FALSE if token has been processed.  A token has been processed
for the following conditions:

            if join -> has been processed if subj mbr GV

            if fail -> has been processed if subj NOT mbr GV

return TRUE otherwise.

NOTE: it is not necessary to detect all possible outcomes... the token is
checked for prior processing ONLY if it is in the external token pool and not
the local token pool.  This condition can only occur if: the token has never
been seen at this member, or the token has been deleted.  If the token has been
deleted, the corresponding comit must have occured.  Thus, check only the final
result as intermediate stages do not effect the local token pool.

***************************************************************************/
int first_time(token, gv, st)
char *token, *gv, *st;
{
  int token_type, rtnval = TRUE;
  char *tmptkn, *subject;
  link *tlist;

  tmptkn = CALLOC(strlen(token) + 1, char);
  strcpy(tmptkn, token);
  tlist = str2list(tmptkn, " #");

  if ( getfromlist(tlist, &subject, 2) == 0 ) {
    printf("\tfirst_time:  parsing failed token subj.\n");
    printf("\ttoken = |%s\n", token);
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$s$$$$$$$$$$$$$$$$\n");
    exit(-1); }


  /* get the token type */
  token_type = GetTokenType(token);

  switch (token_type) {
   case JOINRQSTT:
   case JOINAGREE:
   case JOINCOMIT:
     if (GetRank(gv, subject) != -1) /* subj in GV */
       rtnval = FALSE;
      break;

   case FAILAGREE:
   case FAILCOMIT:
     if (GetRank(gv, subject) == -1) /* subj NOT in GV */
       rtnval = FALSE;
     break;

   default: /* error condition */
         printf("ProcessTokenPool: (first_time) invalid tokentype\n");
              printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(1);
  }

  removelist(tlist);
  free(tmptkn,;

  return(rtnval);

} /* end first_time */
```

Table of Contents

```
/*************************************************************************
* MESSAGE HANDLING AUXILIARY FUNCTIONS  (msgutil.c)
*
*          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 snmbr, 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);
*          char *get_ext_target(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
*                   David Pezdirtz
*
*  Last revision:   27 Jul 1993
*
*************************************************************************/

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

typedef struct link    link;
```

```
/*************************************************************************
          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);

} /* end str2list */
```

```
/**********************************************************************
        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);

          } /* end while list */

      } /* end if len */

    } /* end if list */

  return(header);

} /* end list2str */
```

```
/**********************************************************************
        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);
} /* end removelist */
```

```
/**********************************************************************
        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);

  } /* end else n */

} /* end getfromlist */
```

```
/**********************************************************************
        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);
} /* end listsize */
```

```
/**********************************************************************
        int_2_ext - convert 'inmsg' into an external message.
                'snbr' 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, smbr, orig);
                ...
                free(extmsg);

                If the serial number is not relevant set 'snbr' to 0.
**********************************************************************/
char *int_2_ext(inmsg, smbr, orig)
char *inmsg;
int smbr;
char    *orig;

{
  int     msgsize;
  char    temp[HEADERSIZE];
  char    *extmsg;

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

} /* end int_2_ext */
```

```
/*************************************************************
        get_sr_nbr - extracts the serial number of a message that
                was previously retreived 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, *smbrstr;
  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, &smbrstr, 1) != 1) {  /* get 1st field */
    printf("get_sr_nbr error\n");
    removelist(list);
    free(tmp);
    return(-1); }

  removelist(list);
  free(tmp);
  return(atoi(smbrstr));

} /* end get_sr_nbr */
```

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

        msg_type - returns the integer value correponding to
                the type of the string 'type', as defined in gmp.h
**************************************************************/
int     msg_type(type)
char    *type;
{
 int msgtype = INVALDMSG;

 if (strcmp(type,"tokentokn") == 0)
   msgtype = TOKENTOKN;
 if (strcmp(type,"tokenpool") == 0)
   msgtype = TOKENPOOL;
 if (strcmp(type,"tokenackn") == 0)
   msgtype = TOKENACKN;
 if (strcmp(type,"statusqry") == 0)
   msgtype = STATUSQRY;
 if (strcmp(type,"statusrpt") == 0)
   msgtype = STATUSRPT;
 if (strcmp(type,"statustbl") == 0)
   msgtype = STATUSTBL;
 if (strcmp(type,"initparam") == 0)
   msgtype = INITPARAM;
 if (strcmp(type,"joinreqst") == 0)
   msgtype = JOINREQST;
 if (strcmp(type,"updstatus") == 0)
   msgtype = UPDSTATUS;
 if (strcmp(type,"groupview") == 0)
   msgtype = GROUPVIEW;
 if (strcmp(type,"updatview") == 0)
   msgtype = UPDATVIEW;
 if (strcmp(type,"sndinipar") == 0)
   msgtype = SNDINIPAR;
 if (strcmp(type,"inittoken") == 0)
   msgtype = INITTOKEN;
 if (strcmp(type,"viewreqst") == 0)
   msgtype = VIEWREQST;
 if (strcmp(type,"statreqst") == 0)
   msgtype = STATREQST;
 if (strcmp(type,"tokpreqst") == 0)
   msgtype = TOKPREQST;
 if (strcmp(type,"initgview") == 0)
   msgtype = INITGVIEW;
 if (strcmp(type,"inittable") == 0)
   msgtype = INITTABLE;
 if (strcmp(type,"inittpool") == 0)
   msgtype = INITTPOOL;
 if (strcmp(type,"timeout__") == 0)
   msgtype = TIMEOUT__;
 if (strcmp(type,"starttimr") == 0)
   msgtype = STARTTIMR;
 if (strcmp(type,"delttoken") == 0)
   msgtype = DELTTOKEN;
 if (strcmp(type,"extknpool") == 0)
   msgtype = EXTKNPOOL;

 return(msgtype);

} /* end msg_type */
```

```
/************************************************************
        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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    removelist(list);
    free(tmp);
    return(-1); }

  msgtype = msg_type(type);
  removelist(list);
  free(tmp);
  return(msgtype);

} /* end in_msg_type */
```

```
/************************************************************
        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.
************************************************************/
in' 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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    removelist(list);
    free(tmp);
    return(-1); }

  msgtype = msg_type(type);
  removelist(list);
  free(tmp);
  return(msgtype);

} /* end ext_msg_type */
```

```
/*****************************************************************
        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");
    printf("get_target : string is !%s\n", str);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  nbr = CALLOC(strlen(nbrtmp) + 1, char);
  strcpy(nbr, nbrtmp);

  removelist(list);
  free(tmp);

  return (nbr);
} /* end get_target */
```

```
/*****************************************************************
        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, *tnbr, *tmp;
  link      *list;

  tmp = CALLOC(strlen(str) + 1, char);
  strcpy(tmp,str);
  list = str2list(tmp,"\n ");

  if ( getfromlist(list, &tnbr, 2) != 2) {
    printf("get_originator error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  nbr = CALLOC(strlen(tnbr) + 1, char);
  strcpy(nbr,tnbr);

  removelist(list);
  free(tmp);
  return (nbr);

} /* end get_originator */
```

```
/****************************************************************
            get_ext_target - extracts the destination field 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_ext_target(msg);
                        ...
                        free(nbr);
****************************************************************/
char *get_ext_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, 4) != 4) {
    printf("get_ext_target error\n");
    printf("get_ext_target : string is |%s\n", str);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  nbr = CALLOC(strlen(nbrtmp) + 1, char);
  strcpy(nbr, nbrtmp);
  removelist(list);

  free(tmp);
  return (nbr);

} /* end get_ext_target */
```

Table of Contents

```
/***************************************************************************
* 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");
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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");
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$^$$$\n");
    exit(1); }

  if ( getsockname(sockfd, (struct sockaddr *) &sin, &sinlen )< 0) {
    printf("\tcreateUDP: can't bind local address \n");
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

  *port = sin.sin_port;

  return sockfd;

} /* end createUDP */
```

```
/********************************************************************
            createUN - establish an Unix Domain socket for a server
********************************************************************/
int createUN(path)
char*       path;

{
  struct sockaddr_un            sunx;      /* 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");
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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");
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

  return sockfd;

} /* end createUN */
```

```
/********************************************************************
            connectUN - establish an Unix Domain socket for a client
********************************************************************/
int connectUN(server_path)
char*       server_path;

{
  struct sockaddr_un            sunx;      /* 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.  Attempted !%s\n", server_path);
    printf("\t        ERROR CODE = %d\n", errno);
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

  /* Connect to the server */
  if ( connect(sockfd, (struct sockaddr *) &sunx, sunlen )< 0) {
    printf("\tconnectUN: can't connect to unix server \n");
    printf("\07\t$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

  return sockfd;

} /* end connectUN */
```

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

{
  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;

  } /* end while */

  return(nbytes - nleft);

} /* end readn */
```

```
/*******************************************************************
        writen - Write "n" bytes to a descriptor
                Use in place of wite() when fd is a stream socket
*******************************************************************/
int writen(fd, ptr, nbytes)
register int        fd;
register char       *ptr;
register int        nbytes;

{
  int      nleft, nwriten;

  nleft = nbytes;

  while (nleft > 0) {
    nwriten = write(fd, ptr, nleft);

    if ( nwriten < 0)
           return(nwriten);

    nleft -= nwriten;
    ptr   += nwriten;

  } /* end while */

  return(nbytes - nleft);

} /* end writen */
```

```
/*********************************************************************
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 int        fd;
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 */

   } /* end for */

 return(n);

} /* end readmsg */
```

```
/**********************************************************************
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 int        fd;
register char       *ptr;
register int        n;

{
  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));

} /* end writemsg */
```

```
/****************************************************************
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. T.is 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 ll%sll 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)))
    .. = 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);

} /* end senmsg */
```

```
/*****************************************************************
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 int            fd;
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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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);

} /* end recmsg */
```

A SIMPLE APPLICATION
and
MAIN PROCESS

```
/************************************************************************
* GROUP MEMBERSHIP PROTOCOL - SIMPLE APPLICATION
*
*           This is an example applicaion, 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()
************************************************************************
*  Writen by:       Fernando J. Pire
*  Last revision:   9 Mar 1993
************************************************************************/

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

void killelmnt();

void main(argc,argv)

int       argc;
char      *argv[];

{
  int      aplsocun, newsoc, childpid, clen, msglen;
  char     *aplpath, *groupname, *sitelist, *msg;
  struct   sockaddr_un          caller_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);

  } /* end switch */


/*****************************************************************
          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, killelmnt);
*********************************************************
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, &clen)) >= 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);

  } /* end while */

 printf("APPLICATION: accept error\n");

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

} /* end simpleapp */
```

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

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

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

} /* end killelmnt */
```

```
/*********************************************************************
* GROUP MEMBERSHIP PROTOCOL - MAIN PROCESS
*
*          This program is executed by the application, and spawns
* complete implementation of an element running the Membership Protocol
*          This program waits for a child to cease execution (meanin
* that the element has or is to cease existence, and then releases all
* resources to the operanting 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 fron\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'.
*
*********************************************************************
```

```
*********************************************************************
*  Writen by:      Fernando J. Pires
*  Last revision:  9 Mar 1993
*
*********************************************************************/
#include "gmp.h"
#include "socutil.c"

void killelmnt();

/* these variables are common to 'mainproc` and 'killelmnt` */
char      *fpath, *bpath, *stmonpath, *streppath, *timerpath, *joinppath,
          *intmbrpath, *agrppath, *comppath, *gvmpath, *stmpath,
          *tpmpath, *aplpath,  *grouppathname;

int main(argc,argv)
int       argc;
char      *argv[];


{
  int     fsocudp, fsocun, bsocudp, bsocun, stmonfd, strepfd, timerfd,
          joinpfd, intmbrfd, agrpfd, compfd, gvmfd, stmfd, tpmfd,
          commitpid, frontpid, backpid, gvmpid, stmpid, tpmpid, timerpid,
          streppid, stmonpid, intmbrpid, agreepid, joinpid, returnpid;
  char    sfudp[MAXFD], sbudp[MAXFD], sfun[MAXFD], sbun[MAXFD], sstmon[MAXFD],
          sstrep[MAXFD], stimer[MAXFD], sjoinp[MAXFD], sintmbr[MAXFD],
          sagrp[MAXFD], scomp[MAXFD], sgvm[MAXFD], sstm[MAXFD], stpm[MAXFD],
          sfport[MAXPORT], sbport[MAXPORT], my_name[MAXHOSTNAME + 1],
         *ip_addr, my_addr[MAXLMTSIZE + 1], *groupname, *sitelist;

  u_short  fport, bport;

  struct  hostent  *hptr;
```

```
/*****************************************************
        ESTABLISH A SIGNAL HANDLER TO INTERCEPT
        THE ELEMENT FAILURE SIGNAL FROM APPLICATION
*****************************************************/
 signal(SIGALRM, killelmnt);


/*****************************************************
        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("Sorry 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");
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1);

 } /* end switch */

 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 availabe 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 availabe 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 COMMIT PROCESSOR Unix Domain Stream socket     */
  comppath = UNIXSTR_TMPL;          /* Default path template */
  mktemp(comppath);                 /* Get unique file name  */
  compfd = createUN(comppath);

  if (listen(compfd,5) < 0) {
    printf("\tmainproc: Listen for commit socket failed errno = %d\n", errno);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n"); }

  sprintf(scomp,"%d", compfd);
  printf("COMMIT PROCESSOR: unix socket path => %s\n", comppath);
  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(agrpfd, 5);
  sprintf(sagrp,"%d",agrpfd);
  printf("AGREEMENT PROCESSOR: unix socket path => %s\n", agrppath);
  printf("\n");

/* open a GROUP VIEW MANAGER Unix Domain Stream socket  */
  gvmpath = UNIXSTR_TMPL;          /* D:fault path template */
  mktemp(gvmpath);                 /* Get unique file name  */
  gvmfd = createUN(gvmpath);
  listen(gvmfd, 5);
  sprintf(sgvm,"%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(sstm,"%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 l%sl\n", my_addr);
  printf("\n**************************************************\n\n");

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

/* execute COMMIT PROCESSOR process */
  if ( (commitpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (commitpid == 0) {       /* child process */
          execlp("commit", "commit", scomp, my_addr, stmpath, gvmpath,
                    tpmpath, intmbrpath, fpath, (char*) NULL);
          printf("Error executing comp\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute FRONT process */
  if ( (frontpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (frontpid == 0) {         /* child process */
          execlp("front", "front", sfudp, sfun, my_addr, streppath,
                    joinppath, intmbrpath, (char*)NULL);
          printf("Error executing front\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }
```

```
/* execute BACK process */
  if ( (backpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (backpid == 0) {          /* child process */
          execlp("back", "back", sbudp, sbun, my_addr, stmonpath, agrppath, (char*)NULL);
          printf("Error executing back\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute GROUP VIEW MANAGER process */
  if ( (gvmpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (gvmpid == 0) {          /* child process */
          execlp("gvm", "gvm", sgvm, my_addr, aplpath, grouppathname, (char*) NULL);
          printf("Error executing gvm\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute STATUS TABLE MANAGER process */
  if ( (stmpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (stmpid == 0) {          /* child process */
          execlp("stm", "stm", sstm, (char*) NULL);
          printf("Error executing stm\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute TOKEN POOL MANAGER process */
  if ( (tpmpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (tpmpid == 0) {          /* child process */
          execlp("tpm", "tpm", stpm, (char*) NULL);
          printf("Error executing tpm\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }
```

```
/* execute TIMER process */
  if ( (timerpid = fork()) == -1 )
    printf("Can't fork\n");

  else
    if (timerpid == 0){          /* child process */
          execlp("timer", "timer", stimer, stmonpath, (char*) NULL);
          printf("Error executing timer\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute STATUS REPORTER process */
  if ( (streppid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (streppid == 0) {          /* child process */
          execlp("strep", "strep", sstrep, my_addr, tpmpath, fpath, (char*) NULL);
          printf("Error executing strep\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute STATUS MONITOR process */
  if ( (stmonpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (stmonpid == 0) {          /* child process */
          execlp("stmon", "stmon", sstmon, my_addr, stmpath, gvmpath,
                    agrppath, timerpath, bpath, (char*) NULL);
          printf("Error executing stmon\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

/* execute JOIN PROCESSOR process */
  if ( (joinpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (joinpid == 0){          /* child process */
          execlp("joinp", "joinp", sjoinp, my_addr, stmpath, gvmpath, tpmpath,
          agrppath, intmbrpath, bpath, fpath, grouppathname, sitelist,
                (char*) NULL);
                printf("Error executing joinp\n");
                printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                exit(1); }

/* execute INTEGRATE MEMBER process */
  if ( (intmbrpid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (intmbrpid == 0) {        /* child process */
                execlp("intmbr", "intmbr", sintmbr, my_addr, gvmpath, stmpath,
                        tpmpath, bpath, (char*) NULL);
                printf("Error executing intmbr\n");
                printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                exit(1); }

/* execute AGREEMENT PROCESSOR process */
  if ( (agreepid = fork()) == -1 )
    printf("Can't fork\n");
  else
    if (agreepid == 0) {         /* child process */
                execlp("agree", "agree", sagrp, my_addr, stmpath, gvmpath, tpmpath,
                        joinppath, comppath, fpath, (char*) NULL);
                printf("Error executing agrp\n");
                printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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(agrpfd);
  close(compfd);
```

```
  close(gvmfd);
  close(stmfd);
  close(tpmfd);

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

  printf("\07MAIN PROCESS: ");

  if (returnpid == commitpid)
    printf("Commit");
  if (returnpid == frontpid)
    printf("FRONT");
  if (returnpid == backpid)
    printf("BACK");
  if (returnpid == gvmpid)
    printf("GVM");
  if (returnpid == stmpid)
    printf("STM");
  if (returnpid == tpmpid)
    printf("TPM");
  if (returnpid == timerpid)
    printf("Timer");
  if (returnpid == streppid)
    printf("Status Reporter");
  if (returnpid == stmonpid)
    printf("Status Monitor");
  if (returnpid == intmbrpid)
    printf("Integrate Mbr");
  if (returnpid == agreepid)
    printf("Agree");

  printf(" has returned (pid = %d)\n", returnpid);
  printf("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");

  sleep(120);

  printf("MAIN PROCESS: ");
  if (returnpid == commitpid)
    printf("Commit");
  if (returnpid == frontpid)
    printf("FRONT");
  if (returnpid == backpid)
    printf("BACK");
  if (returnpid == gvmpid)
    printf("GVM");
  if (returnpid == stmpid)
    printf("STM");
  if (returnpid == tpmpid)
    printf("TPM");
  if (returnpid == timerpid)
    printf("Timer");
  if (returnpid == streppid)
    printf("Status Reporter");
  if (returnpid == stmonpid)
    printf("Status Monitor");
  if (returnpid == intmbrpid)
    printf("Integrate Mbr");
  if (returnpid == agreepid)
    printf("Agree");

  printf(" has returned (pid = %d)\n", returnpid);

/* 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);
```

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

} /* end mainproc */


/*********************************************************
        SIGNAL HANDLER THAT CATCHES ELEMENT DEPARTURE
*********************************************************/
void killelmnt()

{
  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);

 } /* end killelmnt */
```

FIFO CHANNEL

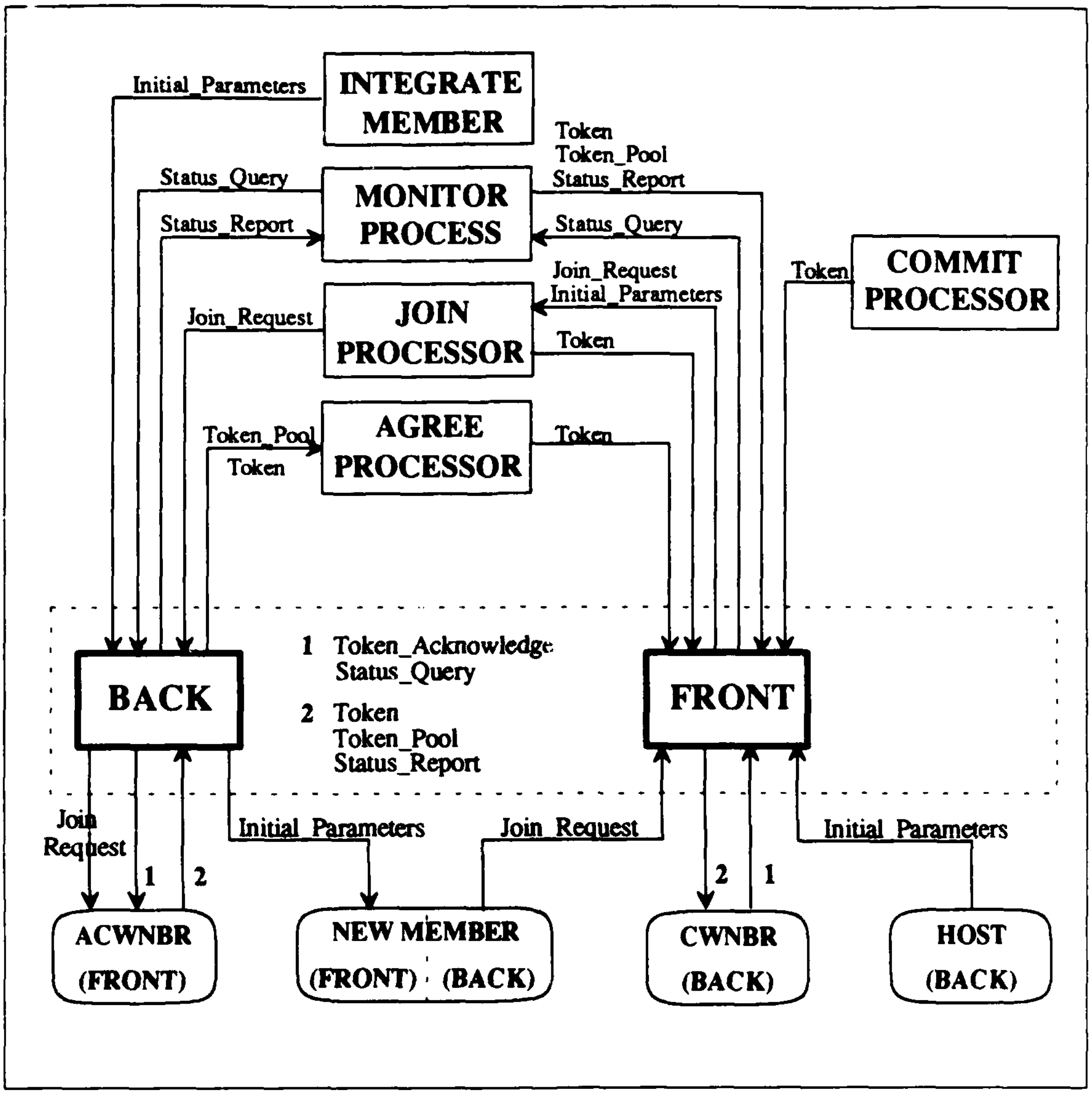

Figure A1 FIFO Channel - Process Dependencies

```
FIFO Channel - FRONT Process

Wait for a channel ready to read
if (external channel ready)
    if (Status_Query)
        send Status_Query to MonitorProcess
    else if (JoinRequest)
        send JoinRequest to JoinProcessor
    else if (InitialParameters)
        send InitialParameters to JoinProcessor
    else if (TokenAck)
        if (Received_Serial_Number = Expected_serial_number)
            remove Head_of_Queue
            decrement Queue_Counter
        end
    end
else /* internal channel ready */
    if (Token)
        change Token to external format  /* add external header */
        insert Token in queue
        increment Serial_Number
        increment Queue_Counter
    else if (TokenPool)
        discard all messages in queue
        change TokenPool to external format  /* add external header */
        insert TokenPool in queue
        increment Serial_Number
        increment Queue_Counter
    else if (StatusReport)
        update cwnbr
        send StatusReport to cwnbr
    end
end
if (Queue_Counter > 0)
    send Head_of_Queue to cwnbr
    set Expected_serial_number = Head_of_Queue_serial_number
end
```

Figure A2 FIFO Channel - Front Process

```
FIFO Channel - BACK process

Wait for a channel ready to ready
if (internal channel ready)
    if (Status_Query)
        update acwnbr
        send Status_Query
    else if (Initial_Parameters)
        update acwnbr
        send Initial_Parameters
    else if (Join_Request)
        send Join_Request
    end
else /* external channel ready */
    if (message originator = acwnbr)
        if (Status_Report)
            send Status_Report to MONITOR_PROCESS
        else if (Token)
            if (Serial_Number = Expected_Serial_Number - 1)
                send Token_Ack /* to acwnbr */
            end
            if (Serial_Number = Expected_Serial_Number )
                send Token to AgreeProcessor
                send Token_Ack /* to acwnbr */
                increment Expected_Serial_Number
            end /* out of order messages are discarded */
        else if (Token_Pool) /* Token_Pool is always accepted */
            send Token_Pool to AgreeProcessor
            send Token_Ack /* to acwnbr */
            set Expected_Serial_Number = Serial_Number + 1
        end
    end
end
```

Figure A3 FIFO Channel - Back Process

```
/*************************************************************************
* FRONT PORT MANAGER                                                     *
*                                                                        *
*  This program is executed by mainproc.                                 *
*                                                                        *
**************************************************************************
*  Writen by:   Fernando J. Pires                                        *
*               David Pezdirtz                                           *
*                                                                        *
*  Last revision:  11 May 1993                                           *
*                                                                        *
*************************************************************************/

#include "gmp.h"
#include "gmputil.c"
#include "fifoutil.c"

int main(argc,argv)
int  argc;
char  *argv[];

{
  int        sockudp, sockun, newsoc;
  int        msgtype, clen, msglen;
  int        queue_counter=0, expectsmbr=1, smbr=1;
  char       *cwnbr, *topmsg, *msg, *extmsg, *tmpmsg;
  char       *my_addr, *strep, *joinp, *intmbr;
  struct     sockaddr_un  uncaller_addr;
  fd_set     fdread;
  link       *head = NULL, *tail = NULL;

  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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

  /* Initialize cwnbr as my_addr */
  cwnbr = CALLOC(strlen(my_addr) + 1,char);
  strcpy(cwnbr,my_addr);

  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");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(1); }

      if((msglen=readmsg(newsoc,&msg,"#")) < 0) {
        printf(" FRONT unix: read error\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(1); }
```

```
msg[msglen]=NULL; /*turn message into string */

/* Determine which message was received */

msgtype = in_msg_type(msg);

switch (msgtype) {
  case EXTKNPOOL:
    flush_queue(&head, &tail);
    queue_counter = 0;


  case TOKENTOKN:
    extmsg = int_2_ext(msg, smbr, my_addr);
    enqueue(&head, &tail, extmsg);

    free(extmsg);
    smbr++;
    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:
    printf("FRONT: invalid message type !%d\n",msgtype);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1);

} /* end switch */

free(msg);
close(newsoc);
```

```
} /* end if then FD_ISSET(sockun, &fdread) */

else {

  if ( (msglen=recmsg(sockudp, &msg)) < 0 ) {
    printf("FRONT internet: receive error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

  msg[msglen]=NULL; /*turn message into string */

  /* Determines which message was received */
  msgtype = ext_msg_type(msg);


  switch (msgtype) {
    case STATUSQRY:
      send_msg_in(msg, strep);
      break;

    case JOINREQST:
    case INITPARAM:
      send_msg_in(msg, joinp);
      break;

    case TOKENACKN:
      if (get_sr_nbr(msg) == expectsmbr) {

        if (queue_counter > 0)
          queue_counter--;

        dequeue(&head, &tail);
      }
      break;
```

```
            default:
              printf("FRONT: invalid message type !%d\n", msgtype);
              printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(1);

          } /* end switch */

          free(msg);

        } /* end else if (FD_ISSET(sockudp, &fdread) */

        if (queue_counter > 0) {
          get_queue_head(head, &tmpmsg);

          topmsg = CALLOC (strlen(tmpmsg) + 1, char);
          strcpy(topmsg,tmpmsg);

          send_msg_back(topmsg, cwnbr);
          expectsnbr = get_sr_nbr(topmsg);

          free(topmsg);

        } /* if (queue_counter > 0) */

      } /* end while TRUE */

    } /* end FRONT */
```

```
/*************************************************************************
* BACK PORT MANAGER                                                      *
*                                                                        *
*         This program is executed by mainproc.                          *
*                                                                        *
**************************************************************************
*  Writen by:  Fernando J. Pires                                         *
*                  David Pezdirtz                                        *
*                                                                        *
*  Last revision:     26 Jul 1993                                        *
*                                                                        *
**************************************************************************/

#include "gmp.h"
#include "gmputil.c"
#include "fifoutil.c"

void       add_originator();

int        main(argc,argv)
int        argc;
char       *argv[];

{
  int        sockudp, sockun, newsoc;
  int        msgtype, clen, msglen;
  int        expectsmbr = 1, smbr, lastsmbr = 0;
  char       *stmon, *agrp, *my_addr, *msg;
  char       *acwnbr, *target, *originator, *extmsg, *last_tp;
  struct     sockaddr_un       uncaller_addr;
  fd_set     fdread;

  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");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(1); }

/* Initialize acwnbr to be my_addr */
acwnbr = CALLOC(strlen(my_addr) + 1,char);
strcpy(acwnbr, my_addr);

/* initialize last token pool to dummy */
last_tp = CALLOC( 6, char);
strcpy(last_tp, "dummy");

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");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\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");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(1); }
```

```
if((msglen=recmsg(newsoc, &msg)) < 0) {
    printf(" BACK unix: read error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(1); }

msg[msglen] = NULL; /*turn message into string */

/* Determine which message was received */
msgtype = in_msg_type(msg);

switch (msgtype) {

    case STATUSQRY:
    case INITPARAM:
          free(acwnbr);
          acwnbr = get_target(msg);

    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:
          printf("BACK: received invalid message type\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(1); }

  free(msg);
  close(newsoc);

} /* end if (FD_ISSET(sockun, &fdread)) */

else {
```

```
if ( (msglen=recmsg(sockudp,&msg)) < 0 ) {
   printf("BACK internet: receive error\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(1); }

msg[msglen] = NULL; /*turn message into string */

originator = get_originator(msg);

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

  /* Determines which message was received */
  msgtype = ext_msg_type(msg);

  switch (msgtype) {
    case STATUSRPT:
        send_msg_in(msg, stmon);
        break;

    case TOKENTOKN:
        srnbr = get_sr_nbr(msg);

        /* check for retransmit of previous token */
        if( srnbr == lastsrnbr){
          send_ack(msg, my_addr, acwnbr);
        }

        /* check for expected message */
        if(expectsrnbr == srnbr) {
          send_msg_in(msg, agrp);
          send_ack(msg, my_addr, acwnbr);
          lastsrnbr = expectsrnbr;
         expectsrnbr++;
        }

        break;
        case EXTKNPOOL:
                send_msg_in(msg, agrp);
                send_ack(msg, my_addr, acwnbr);

                lastsmbr = get_sr_nbr(msg);
                expectsmbr =  lastsmbr + 1;
                break;

        default:
                printf("BACK: received invalid message type\n");
                printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                exit(1);

       } /* end switch */

      } /* end if (strcmp(acwnbr,originator) */

      free(originator);
      free(msg);

   } /* end if (FD_ISSET(sockudp, &fdread)) */

  } /* end while */

} /* end back */
```

MONITOR PROCESS

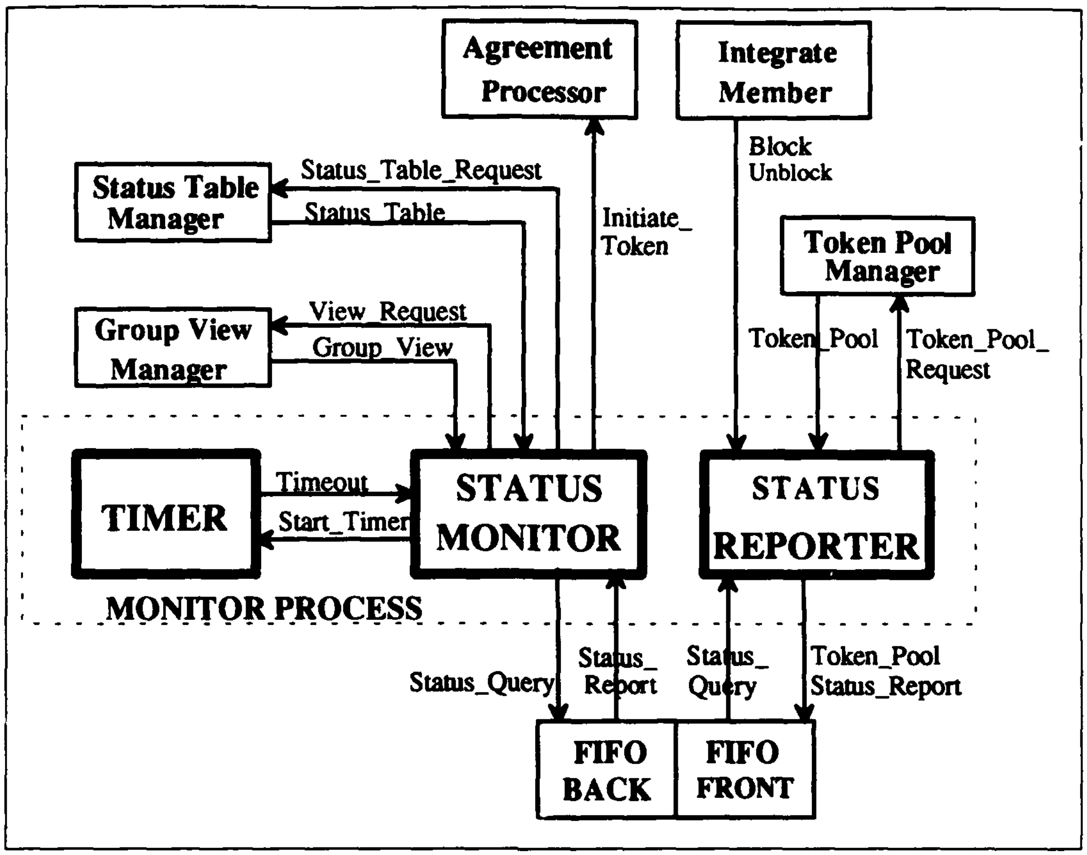

Figure A4 Monitor Process - Internal Structure and Dependencies

```
ReportStatus process at $p_i$
if (not blocked by IntegrateMember)
    if (querying member $\in GV_{p_i}$ or has joinagree status)
        $p_{mon}$ = querying member
        send status to $p_{mon}$
        if (previous querying member = $p_{mon}$)
            send TokenPool($p_i$) to $p_{mon}$
        end
    end
end
end ReportStatus
```

Figure A5 Reporting of Status

```
/*****************************************************************************
 * StatusMonitor:
 * Algorithm:   Fig. 3 of the gmp paper
 * Description: Uses one Unix domain socket. Receives socket names
 *              for BACK, status table manager, group view manager, agreement
 *              process, and timer process.
 *
 * Written by:  Shridhar Shukla
 * Date:        10 Mar. 1993
 *****************************************************************************/

#include "gmputil.c"

int GetAcwnbr();
void SendTkn2Agr();

main(argc,argv)
int      argc;
char     *argv[];

{
  int      soc, newsoc, clen, msglen, backfd, timfd, msgtype;
  char     *backsoc, *stmsoc, *gvmsoc, *agrsoc, *timsoc, *myaddr,
           acwnbr[MAXLMTSIZE+1], *query, starttimer[TIMERMSG+1], *msgheader,
           *originator, *timermsgtype, msg2timer[TIMERMSG+1], *buf;
  link     *msg, *msgfromtimer;

  struct sockaddr_un          caller_addr;

   printf("\tStatusMonitor Process: start execution.\n");

  /*
   * Get socket file descriptors from command line argument. myaddr is in
   * the defined messageformat (ipaddr;front;back)
   */
  if (argc == 8){
    soc    = atoi(argv[1]);
    myaddr = argv[2];
    stmsoc = argv[3];
```

```
    gvmsoc = argv[4];
    agrsoc = argv[5];
    timsoc = argv[6];
    backsoc = argv[7]; }
  else{
    printf("\tUsage: StatusReporter soc myaddr stmsoc gvmsoc agrsoc timsoc backsoc\n"):
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /*
   * Send a query, start a timer, wait for a response or a timeout.
   */
  while(TRUE){

    if ( GetAcwnbr(acwnbr, myaddr, stmsoc, gvmsoc) != 0) {
            printf("\tStatusMonitor: acwnbr computation failed.\n");
            printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

    /*assemble a status query*/
    query = CALLOC(QUERYLEN + 1, char);
    strcpy(query, "statusqry");    /*msg type*/
    strcat(query, "\n");           /*separator*/
    strcat(query, acwnbr);         /*target of report*/
    strcat(query, " ");            /*separator*/
    strcat(query, myaddr);         /*originator of report*/
    strcat(query, "#");            /*end of message delimiter*/

    /*send the query*/
    msglen = strlen(query);
    backfd = connectUN(backsoc);

    if ( writemsg(backfd, query, msglen) < msglen) {
            printf("\tStatusMonitor: query send failed.\n");
            printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

    close(backfd);
    free(query);
```

```
/*send start timer message*/
strcpy(starttimer, "starttimr\ntpad#");
msglen = strlen(starttimer);
timfd  = connectUN(timsoc);

if ( writemsg(timfd, starttimer, msglen) < msglen) {
        printf("\tStatusMonitor: start timer failed.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

close(timfd);

/* Accept status report or timeout from back */
clen = sizeof(caller_addr);

if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
        printf("\tStatusMonitor unix: accept error.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

/* Read either a response to the query or timeout message*/
if ( (msglen=readmsg(newsoc,&buf,"#")) < 0 ) {
        printf("\tStatusMonitor: unix PORT read error.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

buf[msglen] = NULL;
close(newsoc);

/*determine the message type. */
msg = str2list(buf, "\n #");

if ( getfromlist(msg, &msgheader, 1) == 0) {
        printf("\tStatusMonitor: msgtype parsing failed.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }
```

```
if ( strcmp(msgheader, "timeout__") == 0){
        msgtype = TIMEOUT__; /* only tpad timeout possible,type not checked */
}
else
        if ( strcmp(msgheader, "statusrpt") == 0) {  /*also identify  sender */

          msgtype = STATUSRPT;

          if ( getfromlist(msg, &originator, 3) == 0 ) {
            printf("\tStatusMonitor: sts rpt parsing failed.\n");
            printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

          if ( strcmp(acwnbr, originator) != 0) {
            printf("\tStatusMonitor: sts rpt not from present acwnbr.\n");
            printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

        } /* end if strcmp(msgheader, "statusrpt") */

removelist(msg);
free(buf);

switch (msgtype) {
        case TIMEOUT__: /* send initiate agreement msg and block */
            SendTkn2Agr(agrsoc, "failagree", acwnbr);
            break;

        case STATUSRPT: /* send start qry timer and wait until its time */
                              /* to send the next query                  */
            /* assemble start timer msg */
            strcpy(msg2timer, "starttimr\ntqry#");
            msglen = strlen(msg2timer);
            timfd  = connectUN(timsoc);

            if ( writemsg(timfd, msg2timer, msglen) < msglen) {
              printf("\tStatusMonitor: qry timr strt failed.\n");
              printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(-1); }
close(timfd);

/* Accept query expiration msg from timer */
clen = sizeof(caller_addr);

if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
  printf("\tStatusMonitor: accept from timer failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

if ( (msglen = readmsg(newsoc, &buf,"#")) < 0 ) {
  printf("\tStatusMonitor: timer read failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

buf[msglen] = NULL;
close(newsoc);

msgfromtimer = str2list(buf, "\n#");

if ( getfromlist(msgfromtimer, &timermsgtype, 2) == 0) {
  printf("\tStatusMonitor: timr msg parsing failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

/*
 * The following discards a tpad msg from timer that is delayed.
 * A two-level if is used instead of a while because there is
 * exactly one tpad to be discarded if any.
 */
if ( strcmp(timermsgtype, "tqry") != 0) {
  removelist(msgfromtimer);
  free(buf);
  clen = sizeof(caller_addr);

    if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
          printf("\tStatusMonitor: accept from timer failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    if ( (msglen = readmsg(newsoc,&buf,"#")) < 0 ) {
          printf("\tStatusMonitor: timer read failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    buf[msglen] = NULL;
    close(newsoc);

    msgfromtimer = str2list(buf, "\n#");

    if ( getfromlist(msgfromtimer, &timermsgtype, 2) == 0) {
          printf("\tStatusMonitor: timr msg parsing failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }
    if ( strcmp(timermsgtype, "tqry") != 0) {
          printf("\tStatusMonitor: timer func failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }
   } /* end if ( strcmp(timermsgtype, "tqry") */
   removelist(msgfromtimer);
   free(buf);
   break;

  default:
   printf("StatusMonitor: received invalid msg type\n");
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1);

  } /* end switch */
 } /* end while TRUE */
} /* end main */
```

```
/***********************************************************************
 * StatusReporter:
 * Algorithm:   Fig. 4 of the gmp paper
 * Description:  Uses one Unix domain socket. Receives FRONT and token pool manager
 *           socket names along with addr of the  member it belongs to.
 * Written by:  Shridhar Shukla
 *              David Pezdirtz
 *
 * Date:     27 Jul 1993
 *
 ***********************************************************************/

#include "gmputil.c"

main(argc,argv)
int       argc;
char      *argv[];

{
  int   soc, newsoc, clen, msglen, frontfd, tpmfd;
  char  *frontsoc, *tpmsoc, *myaddr, mon[MAXLMTSIZE], prev_mon[MAXLMTSIZE],
        *querydata, *buf, *target, *originator, tkpreq[NODATAMSG+1],
        *msg, *pool;
  link   *query, *query_components, *list;

  struct  sockaddr_un          caller_addr;

  printf("\tStatusReporter Process: start execution\n");

  /* Get socket file descriptors from command line argument */
  if (argc == 5){

    soc      = atoi(argv[1]);
    myaddr   = argv[2];
    tpmsoc   = argv[3];
    frontsoc = argv[4]; }

  else{
    printf("\tUsage: StatusReporter soc myaddr tpmsoc frontsoc\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  strcpy(prev_mon, myaddr); /* initially, one monitors self */

/***********************************************************************
  * Wait for a query to appear and respond to it.
  ***********************************************************************/

  while(TRUE){
    clen = sizeof(caller_addr);

    /* Accept status query from front */
    clen = sizeof(caller_addr);

    if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
          printf("StatusReporter unix: accept error.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    /* Read message */
    if ( (msglen=readmsg(newsoc,&buf,"#")) < 0 ) {
          printf("\tStatusReporter: unix PORT read error.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    buf[msglen]=NULL;

    query = str2list(buf, "\n#");

    if ( getfromlist(query, &querydata, 2) == 0){
          printf("\tStatusReporter: query parsing failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

    query_components = str2list(querydata, " ");
```

```
if ( getfromlist(query_components, &target, 1) == 0){
        printf("\tStatusReporter: query component parsing failed.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

if (strcmp(myaddr, target) != 0){
        printf("\tStatusReporter: target of query is not me.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

if ( getfromlist(query_components, &originator, 2) == 0){
        printf("\tStatusReporter: query component parsing failed.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

strcpy(mon, originator);
removelist(query);
removelist(query_components);

/*assemble a status report message in the same place as the query received*/
strcpy(ouf, "statusrpt"); /*msg type*/
strcat(buf, "\n");        /*separator*/
strcat(buf, mon);         /*target of report*/
strcat(buf, " ");         /*separator*/
strcat(buf, myaddr);      /*originator of report*/
strcat(buf, "#");         /*end of message delimiter*/

/*send the status report out*/
msglen  = strlen(buf);
frontfd = connectUN(frontsoc);    /*open connection*/

if ( writemsg(frontfd, buf, msglen) < msglen) {  /*write all the bytes*/
        printf("\tStatusReporter: report send failed.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

close(frontfd);

free(buf); /*storage acquired on read of query returned*/
```

```
if (strcmp(mon, prev_mon) != 0) { /* if there is a new monitor */
        strcpy(prev_mon, mon);    /* update the previous monitor */

        /*get the token pool*/
        strcpy(tkpreq, "tokpreqst#");
        msglen = strlen(tkpreq);
        tpmfd  = connectUN(tpmsoc);

        if ( writemsg(tpmfd, tkpreq, msglen) < msglen ){
            printf("\tStatusReporter: token pool request failed.\n");
            printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

        if ( (msglen=readmsg(tpmfd,&buf,"#")) < 0 ) {
            printf("\tStatusReporter: token pool read failed.\n");
            printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

        buf[msglen] = NULL;
        close(tpmfd);

        /* create the external token pool message */
        list = str2list(buf,"\n#");

        if ( getfromlist(list, &pool, 2) == 0){
            printf("\tStatusReporter: token pool parsing failed.\n");
            printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

        msg = CALLOC(9 + strlen(myaddr) + strlen(pool) + 3 + 1, char);
        strcpy(msg, "extknpool\n");
        strcat(msg, myaddr);
        strcat(msg, "\n");
        strcat(msg, pool);
        strcat(msg, "#");

        /*send the token pool to monitor via front*/
        msglen = strlen(msg);
        frontfd = connectUN(frontsoc);    /*open connection*/
```

```
          if ( writemsg(frontfd, msg, msglen) < msglen ){
              printf("\tStatusReporter: token pool send failed.\n");
              printf("\t07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(-1);}

          close(frontfd);

          removelist(list);
          free(buf);
          free(msg);

      } /* end if (strcmp(mon, prev_mon) != 0) */

      close(newsoc);

    } /* end while(TRUE) */

} /* end StatusReporter */
```

```
/************************************************************************
 * Timer:
 * Algorithm:   when a tpad msg from StatusMonitor (sm, for short) is
 *          received,a timer for an interval equal to the value returned
 *          by GetNextTpad is started.  In between successive ticks of
 *          the timer, the nonblocking Unix domain socket is checked for
 *          a tqry msg from sm.  If one is received before the timer
 *          runs out, the timer value is reset to a value returned by
 *          GetNextTqry.  A tqry timeout msg is sent to sm. If the tpad
 *          timer runs out, a tpad timeout msg is sent.
 *
 * Description:  Uses one NONBLOCKING Unix domain socket. Receives socket name
 *              for sm.
 * Written by:  Shridhar Shukla
 * Date:        22 Feb. 1993
 ************************************************************************/
#include "gmputil.c"

int GetNextTpad();
int GetNextTqry();

main(argc,argv)
int         argc;
char        *argv[];
{
  int         soc, newsoc, clen, msglen, smfd, on, countval, countfortqry;
  char        *smsoc, *timermsgtype, *buf, timeoutmsg[TIMERMSG+1];
  link        *msglist;

  struct  sockaddr_un         caller_addr;

  /* Get socket file descriptors from command line argument. */
  if (argc == 3) {
    soc   = atoi(argv[1]);
    smsoc = argv[2]; }
  else {
    printf("\tUsage: timer soc smsoc\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
while(TRUE) {

  /* Make the following read blocking */
  on = BLOCKING;
  ioctl(soc, FIONBIO, (char *) &on);

  /* Accept a start timer msg. */
  clen = sizeof(caller_addr);

  if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ) {
        printf("\tTimer: unix accept error.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

  if ( (msglen=readmsg(newsoc,&buf,"#")) < 0 ) {
        printf("\tTimer: unix PORT read error.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

  buf[msglen]=NULL;
  close(newsoc);

  /* determine the message type. */
  msglist = str2list(buf, "\n#");

  if ( getfromlist(msglist, &timermsgtype, 2) == 0) {
        printf("\tTimer: msgtype parsing failed.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

  if ( strcmp(timermsgtype, "tpad") == 0) { /* start tpad timer */
        countval = GetNextTpad(); }

  else {  printf("\tTimer: start timer msg not for tpad.\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

  removelist(msglist);
  free(buf);
countfortqry = FALSE;

/* Make the following read nonblocking */
on = NONBLOCKING;
ioctl(soc, FIONBIO, (char *) &on); /* CHECK: is the cast necessary? */

while (CountDown(&countval, COUNT_DOWN_STEP) > 0) {
        if ( countfortqry == FALSE ) { /* need to check for msg */
          clen = sizeof(caller_addr);

          if ( ((newsoc = accept(soc, (struct sockaddr*)
                     &caller_addr, &clen)) == -1) ) {

            if ( (errno == EWOULDBLOCK) || (errno == EINPROGRESS) ) {
                     ; /* no connections present */ }

            else { printf("\tTimer: unix accept error.\n");
                     printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                     exit(-1); }

          } /* end then newsoc  = accept */

          else { /* a connection is present */
             sleep(1);

             if ( (msglen=readmsg(newsoc,&buf,"#")) < 0 ) {
                     printf("\tTimer: unix PORT read error.\n");
                     printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                     exit(-1); }

             buf[msglen]=NULL;
             close(newsoc);

             /* msg was read - check its type to be tqry */

             msglist = str2list(buf, "\n#");

             if ( getfromlist(msglist, &timermsgtype, 2) == 0) {
                     printf("\tTimer: msgtype for tqry parsing failed.\n");
                     printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                     exit(-1); }

             if ( strcmp(timermsgtype, "tqry") != 0) {
                     printf("\tTimer: tqry expected, something else rcvd.\n");
                     printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                     exit(-1); }

             countval = GetNextTqry();
             countfortqry = TRUE;

             removelist(msglist);
             free(buf);
           } /* end else newsoc */
     } /* end if ~countfortqry */
   } /* end while CountDown */

   if (countfortqry == TRUE) { /* timeout msg on tqry to be sent */
     strcpy(timeoutmsg, "timeout__\ntqry#");
   }

   else  /* timeout msg on tpad to be sent */
     strcpy(timeoutmsg, "timeout__\ntpad#");

   smfd = connectUN(smsoc);
   msglen = strlen(timeoutmsg);

   if ( writemsg(smfd, timeoutmsg, msglen) < msglen) {
     printf("\tTimer: Timeout msg send failed.\n");
     printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

   close(smfd);

 } /* end while true */

} /* end Timer */
```

```
/***************************************************************************
GetNextTqry(): The count down loop in Timer.c does not attempt to read the
          socket while it is counting down from the value returned by
          this function.
***************************************************************************/
int GetNextTqry()
{
  int       next;

  next = TQRY_INTERVAL;
  return(next);

} /* end GetNextTqry */
```

```
/***************************************************************************
GetNextTpad(): The count down loop in Timer.c attempts to reads the
          socket while it is counting down from the value returned by
          this function. If a msg is read, the down counter is initialized
                to the value returned by GetNextTqry().
***************************************************************************/
int GetNextTpad()
{
  int       next;

  next = TPAD_INTERVAL;
  return(next);

} /* end GetNextTpad */
```

AGREEMENT PROCESSOR

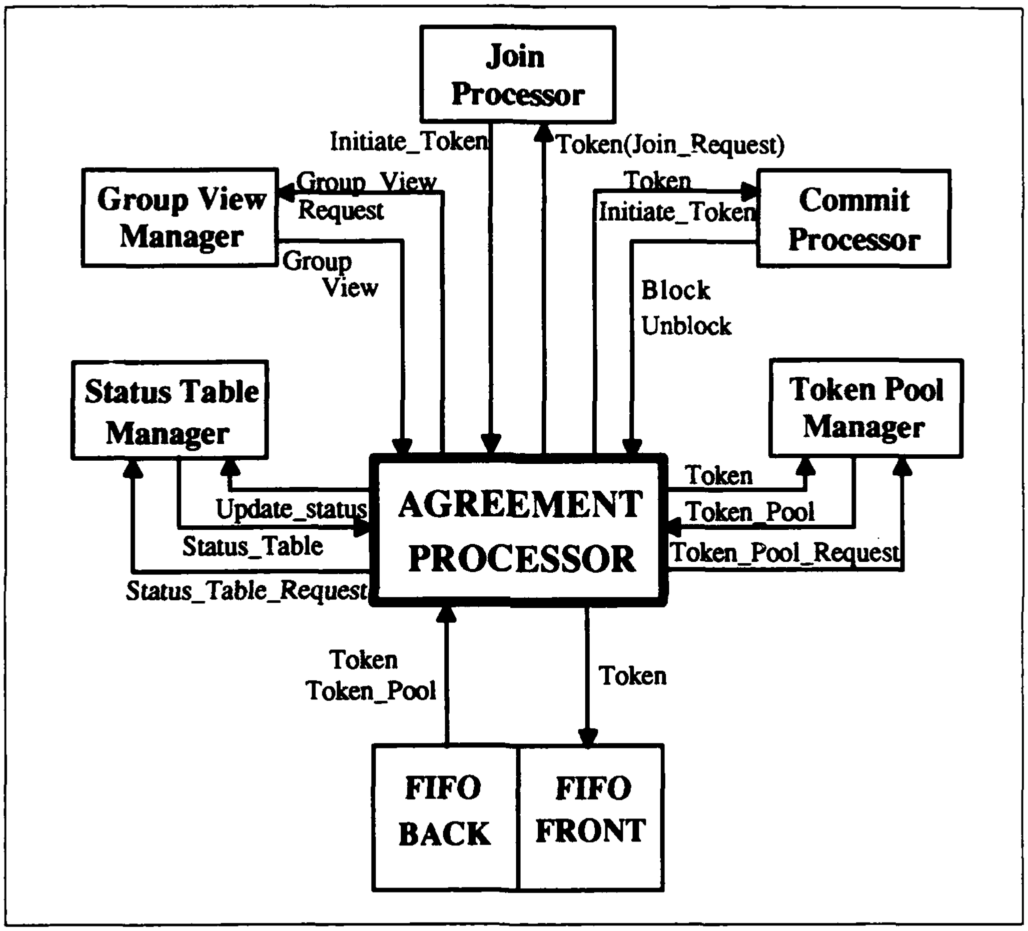

Figure A6 Agreement Processor - Process Dependencies

```
AgreeProcessor for $agree_{p_j}(p_k)$ at $p_i$
if (not blocked by CommitProcessor)
  if (initiate agreement message received)  /* $p_i = p_j$ */
    add $agree_{p_j}(p_k)$ to $TokenPool(p_i)$
    $ST_{p_i}(p_k) \leftarrow$ joinagreed or failagreed
    send $agree_{p_j}(p_k)$ to $cwnbr(p_i)$
    send acknowledgment to calling process
  else  /* a token or external token pool is received */
    if (ExtTokenPool)
      for $\forall$tokens $\in$ ExtTokenPool
        if (token $\in$ $TokenPool(p_i)$)
          if (originator failed)
            ProcessToken
          end
        else /*  token not in TokenPool  */
          if (received for the first time)
            ProcessToken
          end
        end
      end
    else  /* a token was received */
      if (received for the first time)
        ProcessToken
      end
    end
  end
end
```

Figure A7 Agreement Processor

```
LostAgreeToken
if (joinagree)
    if ($rank(p_j) > rank(p_i)$)
        return true
    else
        return false
    end
end

if (failagree)
    if ($RelativeRank(p_k, p_i) > RelativeRank(p_j, p_i)$)
        return true
    else
        return false
    end
end
```

Figure A8 Determination of Token Originator's Failure

```
ProcessToken
if (joinreqst)
      send token to JoinProcessor
elseif (commit)
   send token to ComitProcessor
elseif (agree)
   if (($p_i \neq p_j$) && (agree token $\notin$ TokenPool($p_i$))
      add agree$_{p_j}$($p_k$) to TokenPool($p_i$)
      $ST_{p_i}$($p_k$) $\leftarrow$ FailAgreed or JoinAgreed
      send agree$_{p_j}$($p_k$) to cwnbr($p_i$)
    else                    $p_j$
      if (($p_i = p_j$) || ($\forall p_l \mid p_l \rightarrow p_i, p_l \in ST_{p_i}$))
         compute rank $\forall p_l \in ST_{p_i}$ with Agreed status
         if rank($p_k$) = smallest
            send initiate_comit to ComitProcessor
          else
             $ST_{p_i}$($p_k$) $\leftarrow$ joinpendg or failpendg
         end
      end
   end
end
```

Figure A9 Processing Agree Tokens

```
/*************************************************************************
 * Agreement:
 * Description:  Refer to Fernando's thesis, Chapter 3
 * Written by:   Shridhar Shukla
 *               David Pezdirtz
 *
 * Date:      14 Dec 1993
 *************************************************************************/

#include "gmputil.c"

void IntegrateToken();
void ProcessTokenPool();
void ProcessToken();
void ExecuteAgreement();
int  OriginatorFailed();

main(argc,argv)
int        argc;
char       *argv[];

{
  int     soc, newsoc, clen, msglen, msgtype,
          frontfd;
  char    *myaddr, *stmsoc, *gvmsoc, *tpmsoc, *jpsoc, *comsoc,
          *frontsoc, *buf, token[TOKENMSGLEN+1], *tkn,
          c[1], *inittknmsg;

char *gv, *st;

  struct  sockaddr_un   caller_addr;
  link    *msglist, *tknl;

printf("\tAgreement Process: start execution.\n");
```

```
/*************************************************************************
 * Get socket descriptors from command line argument. myaddr is  (ipaddr;front;back)
 *************************************************************************/

 if (argc != 9) {
   printf("\tUsage: Agreement soc myaddr stmsoc gvmsoc tpmsoc jpsoc comsoc frontsoc \n");
   printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

 else {
  soc      = atoi(argv[1]);
  myaddr   = argv[2];
  stmsoc   = argv[3];
  gvmsoc   = argv[4];
  tpmsoc   = argv[5];
  jpsoc    = argv[6];
  comsoc   = argv[7];
  frontsoc = argv[8]; }

 while (TRUE) {

  /* read msg on local socket */
  clen = sizeof(caller_addr);

  if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
    printf("\tAgreement: unix accept error for incoming msg.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( (msglen=readmsg(newsoc,&buf,"#")) < 0 ) {
    printf("\tAgreement: unix read error for incoming msg.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  buf[msglen] = NULL;

  msgtype = in_msg_type(buf);
/* create agree token if initiate token: type not checked yet. */
if ( msgtype == INITTOKEN ) {

  /* assemble token */
  msglist = str2list(buf, "\n#");

  if ( getfromlist(msglist, &inittknmsg, 2) == 0 ) {
    printf("\tAgreement:  parsing failed for initiate token msg.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  strcpy(token, "tokentokn\n");
  strcat(token, inittknmsg);
  strcat(token, " ");
  strcat(token, myaddr);
  strcat(token, "#");
  removelist(msglist);

  IntegrateToken(token, tpmsoc, stmsoc);     /* update local state */

  /* send token to front */
  msglen  = strlen(token);
  frontfd = connectUN(frontsoc);

  if ( writemsg(frontfd, token, msglen) < msglen) {
    printf("\tAgreement: token send to  front failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(frontfd);

  /* send NULL ack to process requesting initiate token msg */
c[0]   = NULL;
  msglen = strlen(c);

  if ( writemsg(newsoc, c, msglen) < msglen) {
    printf("\tAgreement: ack send to token initiator failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

   } /* end if INITTOKEN */
   else {  /* a token is received */
     switch (msgtype) {
       case EXTKNPOOL:

           ProcessTokenPool(buf, myaddr, tpmsoc, stmsoc, gvmsoc, jpsoc, comsoc, frontsoc);
           break;

       case TOKENTOKN:
            tknl = str2list(buf, "\n#"); /* remove the msg header */

            if ( getfromlist(tknl, &tkn, 2) == 0 ) {
             printf("\tAgreement: parsing failed for a token msg.\n");
             printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(-1); }

            /* get local group view */
            gv = GetGroupView(gvmsoc);

            /* get status table */
            st = GetStatusTable(stmsoc);

            if (first_time(tkn, gv, st)){
              ProcessToken(tkn, myaddr, tpmsoc, stmsoc, gvmsoc, jpsoc, comsoc, frontsoc);
               free(gv);
               free(st);
            }

           removelist(tknl);

       } /* end switch */

     } /* end else */

   close(newsoc);
   free(buf);

  } /* end while (infinite loop) */
} /* end main */
```

```
/*************************************************************************
ProcessTokenPool:

*************************************************************************/
void ProcessTokenPool(tpmsg,myaddr,tpmsoc,stmsoc,gvmsoc,jpsoc,comsoc,frontsoc)
char *tpmsg, *myaddr, *tpmsoc, *stmsoc, *gvmsoc, *jpsoc, *comsoc, *frontsoc;

{
  int     msglen, tpmfd, tpsize, i, gvmfd, process;
  char    *tpsizestrng, tkpreq[NODATAMSG + 1], *ltpmsg, *nexttkn, *tmptp,
          *tp_origin, *t_origin, *gvreq[NODATAMSG + 1], *gv, *st;
  link    *tplist;

  /* break out the individual tokens */
  tmptp = CALLOC(strlen(tpmsg) + 1, char);
  strcpy(tmptp, tpmsg);
  tplist = str2list(tmptp, "\n=#");

  if ( getfromlist(tplist, &tp_origin, 2) == 0) {
    printf("\tProcessTokenPool: token pool msg parsing for size failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(tplist, &tpsizestrng, 3) == 0) {
    printf("\tProcessTokenPool: token pool msg parsing for size failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  tpsize = atoi(tpsizestrng);

  /* get local token pool */
  ltpmsg = GetTokenPool(tpmsoc);

  /* get local group view */
  gv = GetGroupView(gvmsoc);

  /* get status table */
  st = GetStatusTable(stmsoc);

 for (i = 4; i <= tpsize + 3; i++) { /* for all tokens in ext tplist */

  if ( getfromlist(tplist, &nexttkn, i) == 0) {
    printf("\tProcessTokenPool: token pool msg parsing for token %d failed.\n", (i - 3));
    printf("\tExTknPool = |%s|\n", tpmsg);
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  process = FALSE;

  /*
   * process the token from the token pool on the following conditions:
   *     a) in local token pool & the originator has failed
   *     b) not in local token pool & i haven't seen the token before
   */

  if (InTokenPool(ltpmsg, nexttkn)){
   if (OriginatorFailed(nexttkn, gv, myaddr, tp_origin))
      process = TRUE;
  }
  else{ /* token !InTokenPool */

      if (first_time(nexttkn, gv, st))
         process = TRUE;
  }

  if (process){
   ProcessToken(nexttkn, myaddr, tpmsoc, stmsoc, gvmsoc, jpsoc, comsoc, frontsoc);
  } /* end if process */

 } /* end for i */

 removelist(tplist);
 free(tmptp);
 free(ltpmsg);
 free(gv);
 free(st);

} /* end ProcessTokenPool */
```

```
/***************************************************************************
ProcessToken:

****************************************************************************/
void ProcessToken(tkn, myaddr, tpmsoc, stmsoc, gvmsoc, jpsoc, comsoc, frontsoc)
char *tkn, *myaddr, *tpmsoc, *stmsoc, *gvmsoc, *jpsoc, *comsoc, *frontsoc;

{
  link     *tknl;
  char     *subject, *stbuf, *origin, *gv, *st, ltkn[TOKENMSGLEN + 1],
            tknmsg[TOKENMSGLEN + 1],request[NODATAMSG + 1],
            status[STSTYPELEN + 1], *msg;
  int       tkntype, msglen, comfd, stmsglen, stmfd, jpfd, flag;

  strcpy(ltkn, tkn);
  tknl = str2list(ltkn, " ");

  if ( getfromlist(tknl, &subject, 2) == 0 ) {
    printf("\tProcessToken: parsing failed for token subject.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(tknl, &origin, 3) == 0 ) {
     printf("\tProcessToken: parsing failed for token orignator.\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  tkntype = GetTokenType(tkn);

  switch(tkntype) {
    case FAILCOMIT: ; /* processing same for comit tokens */
    case JOINCOMIT:

      strcpy(tknmsg, "tokentokn\n"); /* send token to commit process */
      strcat(tknmsg, tkn);
      strcat(tknmsg, "#");

       msglen = strlen(tknmsg);
      comfd  = connectUN(comsoc);

      if ( writemsg(comfd, tknmsg, msglen) < msglen) {
        printf("\tProcessToken: tkn send to commit process failed.\n");
        printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      /* wait for response back from commit processor */

      if ( (msglen=readmsg(comfd,&msg,"#")) < 0 ) {
        printf("\tExecuteAgreement: fail on response from commit\n");
        printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      msg[msglen] = NULL;

      /* don't check message as only possible is completion flag from commit */
      free(msg);

      close(comfd);
      break;

    case JOINRQSTT:

      if (InGroup(gvmsoc, subject) == -1) { /* subject is NOT in gv */
         /* get the status table */
        stbuf = GetStatusTable(stmsoc);

       /* and subject is NOT in st */
       if (InStatusTable(stbuf, subject, (char *) NULL) == FALSE) {

         /*send token to join process*/
         strcpy(tknmsg, "tokentokn\n");
         strcat(tknmsg, tkn);
         strcat(tknmsg, "#");

         msglen = strlen(tknmsg);
         jpfd   = connectUN(jpsoc);
```

```
          if ( writemsg(jpfd, tknmsg, msglen) < msglen) {
             printf("\tProcessToken: tkn send to join process failed.\n");
             printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
              exit(-1); }

          close(jpfd);

        } /* end if InStatusTable */

        free(stbuf);

      } /* end if subject is NOT in gv */

      break;

    case FAILAGREE:
    case JOINAGREE:

      ExecuteAgreement(tkn,myaddr,tpmsoc,stmsoc,gvmsoc,comsoc,frontsoc);
      break;

    default: /* error condition */
       printf("ProcessToken: invalid tokentype\n");
       printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
       exit(1);

  } /* end switch */

  removelist(tknl);

} /* end ProcessToken */
```

```
/*****************************************************************************
ExecuteAgreement:

*****************************************************************************/
void ExecuteAgreement(token, myaddr, tpmsoc, stmsoc, gvmsoc, comsoc, frontsoc)
char *token, *myaddr, *tpmsoc, *stmsoc, *gvmsoc, *comsoc, *frontsoc;

{
  int     msglen, tpmfd, stmfd, tkntype, duplicate, initcommitphase, maxrank,
          myrank, subjrank, nextrank, i, tpsize, comfd, gvmfd, frontfd,
          smallest, entryrank;
  char  tkpreq[NODATAMSG+1], stmreq[NODATAMSG+1], *tpbuf, *stbuf, *gvbuf,
         *subj, *orig, ltkn[TOKENMSGLEN+1], tknmsg[TOKENMSGLEN+1], *msg,
          request[NODATAMSG+1], *member, *tpsizestrng, *tptkn, *tptknsubj,
          status[STSTYPELEN+1], initcommsg[INITTOKENMSGLEN+1],
          updtstsmsg[UPDTSTSMSGLEN+1];
  link    *tknl, *tpl, *tptknl;

  /* get the token pool */
  tpbuf = GetTokenPool(tpmsoc);

  /* get the status table */
  stbuf = GetStatusTable(stmsoc);

  /* get the group view */
  gvbuf = GetGroupView(gvmsoc);


  /* get the subject originator, subject and tokentype */
  strcpy(ltkn, token);
  tkntype = GetTokenType(ltkn);
  tknl    = str2list(ltkn, " ");

  if ( getfromlist(tknl, &subj, 2) == 0 ) {
     printf("\tExecuteAgreement: parsing failed for token subject.\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }
```

```
if ( getfromlist(tknl, &orig, 3) == 0 ) {
   printf("\tExecuteAgreement: parsing failed for token originator.\n");
   printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

if ( InTokenPool(tpbuf, token) == TRUE) {
  duplicate = TRUE; }
else {
  duplicate = FALSE; }

initcommitphase = FALSE;

/* if I'm not originator & !duplicate */
if ( (strcmp(orig, myaddr) != 0) && (duplicate == FALSE) ) {

  /* assemble token msg */
  strcpy(tknmsg, "tokentokn\n");
  strcat(tknmsg, token);
  strcat(tknmsg, "#");

  IntegrateToken(tknmsg, tpmsoc, stmsoc);

  /* send token to front */
  msglen  = strlen(tknmsg);
  frontfd = connectUN(frontsoc);

  if ( writemsg(frontfd, tknmsg, msglen) < msglen) {
     printf("\tExecuteAgreement: token send to  front failed\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  close(frontfd);

} /* end !originator & !duplicate */

else { /* originator || duplicate */
```

```
/* I'm originator */
if (strcmp(orig, myaddr) == 0) {
  initcommitphase = TRUE;
}
else { /* !originator */

 /***************************************************************
  * decide not to initiate commit if there is at least one operational
  * member between the token subject and myself along the direction
  * of token circulation.
  ***************************************************************/

  initcommitphase = TRUE;

  myrank   = GetRank(gvbuf, myaddr);
  subjrank = GetRank(gvbuf, subj);
  maxrank  = GetGroupSize(gvbuf) - 1;

  if (subjrank == maxrank)
    nextrank = 0;
  else
    nextrank = subjrank + 1;

  while ( nextrank != myrank ) {

    if ( GetMembWithRank(gvbuf, &member, nextrank) == -1 ) {
       printf("\tExecuteAgreement: member with rank %d not there.\n", nextrank);
       printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
       exit(-1); }

    if ( InStatusTable(stbuf, member, (char *) NULL) == FALSE) {
       initcommitphase = FALSE;
       free(member);
       break; }

    if (nextrank == maxrank)
      nextrank = 0;
    else
      nextrank++;
      free(member);

    } /* end while nextrank */

  } /* end else !originator */

} /* end orginator || duplicate */

if (initcommitphase == TRUE) {

  smallest = TRUE;

  /* the for loop below for all entries in the token pool with
   * agree status
   */

  tpl = str2list(tpbuf, "\n=#");

  if ( getfromlist(tpl, &tpsizestrng, 2) == 0) {
     printf("\tExecuteAgreement: token pool parsing for size failed.\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  tpsize = atoi(tpsizestrng);

  for (i = 3; i < tpsize+3; i++) {

    if ( getfromlist(tpl, &tptkn, i) == 0) {
       printf("\tExecuteAgreement: token pool parsing for token %d failed.\n", (i-2));
       printf("\t                : token pool size = !%d\n", tpsize);
       printdatabase(tpl, 4);
       printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
       exit(-1); }

    tptknl = str2list(tptkn, " ");

    if ( getfromlist(tptknl, &tptknsubj, 2) == 0) {
       printf("\tExecuteAgreement: token pool parsing for subj of token %d failed.\n", i);
       printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
       exit(-1);
    }

    if ( InStatusTable(stbuf, tptknsubj, status) == TRUE ) {

      if ((strcmp(status,"failagree") == 0) || (strcmp(status, "joinagree") == 0)) {
        subjrank  = GetRank(gvbuf, subj);
        entryrank = GetRank(gvbuf, tptknsubj);

        if ( subjrank > entryrank ) {

          smallest = FALSE;
          removelist(tptknl);
          break;

        } /* end if subjrank > entryrank */

      } /* end if( "agree" ) */

    } /* end if(InstatusTable ... ) */

    removelist(tptknl);

  } /* end for i */

  removelist(tpl);

  if (smallest == TRUE) { /* send initiate token to commit processor */

    strcpy(initcommsg, "inittoken\n");          /* header */

    /* get the smallest token's status */
    InStatusTable(stbuf, subj, status);


    if (strcmp(status, "failagree") == 0)          /* type */
    strcat(initcommsg, "failcomit");
  else
    strcat(initcommsg, "joincomit");

  strcat(initcommsg, " ");                    /* separator */
  strcat(initcommsg, subj);                   /* subject for token */
  strcat(initcommsg, "#");                    /* end of msg */

  /* send initiate comit msg */
  msglen = strlen(initcommsg);
  comfd  = connectUN(comsoc);

  if ( writemsg(comfd, initcommsg, msglen) < msglen) {
     printf("\tExecuteAgreement: init comit send failed for %s.\n", subj);
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  /* ### wait for response back from commit processor */

  if ( (msglen=readmsg(comfd,&msg,"#")) < 0 ) {
     printf("\tExecuteAgreement: fail on response from commit\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

 msg[msglen] = NULL;

  /* don't check message as only possible is completion flag from commit */

  close(comfd);

 }  /* end then (smallest == TRUE) */

else { /* !smallest -> update status with appropriate pending status */

  strcpy(updtstsmsg, "updstatus\n");
  strcat(updtstsmsg, subj);
  strcat(updtstsmsg, " ");

  if (strcmp(status, "joinagree") == 0)
    strcat(updtstsmsg, "joinpendg");
  else if (strcat(updtstsmsg, "failpendg");
  else {
     printf("\tExecuteAgreement: incorrect status !%s\n", status);
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  strcat(updtstsmsg, "#");

  /* send update status message to status table manager */
  msglen = strlen(updtstsmsg);
  stmfd  = connectUN(stmsoc);

  if ( writemsg(stmfd, updtstsmsg, msglen) < msglen) {
    printf("\tExecuteAgreement: status update to stm failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(stmfd);

 } /* end else !smallest */

} /* end if initcommitphase */

removelist(tknl);
free(tpbuf);
free(stbuf);
free(gvbuf);

} /* end ExecuteAgreement */
```

```
/*************************************************************************
IntegrateToken: updates the token pool and the status table for token after adding
        appropriate message types.
*************************************************************************/
void IntegrateToken(tknmsg, tpmsoc, stmsoc)
char *tknmsg, *tpmsoc, *stmsoc;

{
  int      msglen, fd;
  char     updtstsmsg[UPDTSTSMSGLEN + 1], localtoken[TOKENMSGLEN + 1],
           *subject, *tokentype, statustype[STSTYPELEN + 1];
  link     *tokenlist;

  strcpy(localtoken, tknmsg);

  /* send a token msg to token pool manager */
  msglen = strlen(tknmsg);
  fd    = connectUN(tpmsoc);

  if ( writemsg(fd, tknmsg, msglen) < msglen) {
    printf("\tIntegrateToken: token send to token pool manager failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(fd);

  /* assemble update status message */
  tokenlist = str2list(localtoken, "\n #");

  if ( getfromlist(tokenlist, &subject, 3) == 0 ) {
     printf("\tIntegrateToken:  local token copy parsing for subject failed.\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  strcpy(updtstsmsg, "updstatus\n");
  strcat(updtstsmsg, subject);
  strcat(updtstsmsg, " ");

  if ( getfromlist(tokenlist, &tokentype, 2) == 0 ) {
     printf("\tIntegrateToken:  local token copy parsing for tokentype failed.\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  if (strcmp(tokentype, "failagree") == 0)
    strcpy(statustype, "failagree");

  if (strcmp(tokentype, "failcomit") == 0)
    strcpy(statustype, "delstatus");

  if (strcmp(tokentype, "joinagree") == 0)
    strcpy(statustype, "joinagree");

  if (strcmp(tokentype, "joincomit") == 0)
    strcpy(statustype, "delstatus");

  if (strcmp(tokentype, "joinreqst") == 0)
    strcpy(statustype, "joinrqstd");

  strcat(updtstsmsg, statustype);
  strcat(updtstsmsg, "#");

  removelist(tokenlist);

  /* send update status message to status table manager */
  msglen = strlen(updtstsmsg);
  fd    = connectUN(stmsoc);

  if ( writemsg(fd, updtstsmsg, msglen) < msglen) {
     printf("\tIntegrateToken: status update to status table manager failed.\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

  close(fd);

} /* end IntegrateToken */
```

```
/*************************************************************************
OriginatorFailed:

        return TRUE if:

          RelativeRank(tp_origin, myaddr) < RelativeRank(subject(token), myaddr)
          and token is a 'failagree' token

                          OR

          rank(tp_origin) > rank(myaddr) and token is a "joinagree" token

        otherwise return FALSE

*************************************************************************/
int OriginatorFailed(token, gv, myaddr, origin)
char *token, *gv, *myaddr, *origin;
{
  char  *subject, *tmp_tkn;
  link  *list;
  int   failed, rr_o, rr_s, rank_o, myrank, tokentype;

  tmp_tkn = CALLOC(strlen(token) + 1, char);
  strcpy(tmp_tkn, token);
  list = str2list(tmp_tkn, "\n #");

  if ( getfromlist(list, &subject, 2) == 0 ) {
    printf("\tOriginatorFailed: token parsing for subject failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  failed = FALSE;

  tokentype = GetTokenType(token);

  if (tokentype == JOINAGREE){

    if ((rank_o = GetRank(gv, origin)) < 0){
      printf("\tOriginatorFailed: tp originator not in gv.\n");
      printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    if ((myrank = GetRank(gv, myaddr)) < 0){
      printf("\tOriginatorFailed: i'm not in gv.\n");
      printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    if (rank_o > myrank){
      failed = TRUE;
    }

  } /* end if joinagree */

  if (tokentype == FAILAGREE){

    if ((rr_o = RelativeRank(gv, origin, myaddr)) < 0){
      printf("\tOriginatorFailed: tp originator not in gv.\n");
      printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    rr_s = RelativeRank(gv, subject, myaddr);

    if (rr_o < rr_s)
      failed = TRUE;

  } /* end Failagree */

  removelist(list);
  free(tmp_tkn);

  return(failed);

} /* end OriginatorFailed */
```

COMMIT PROCESSOR

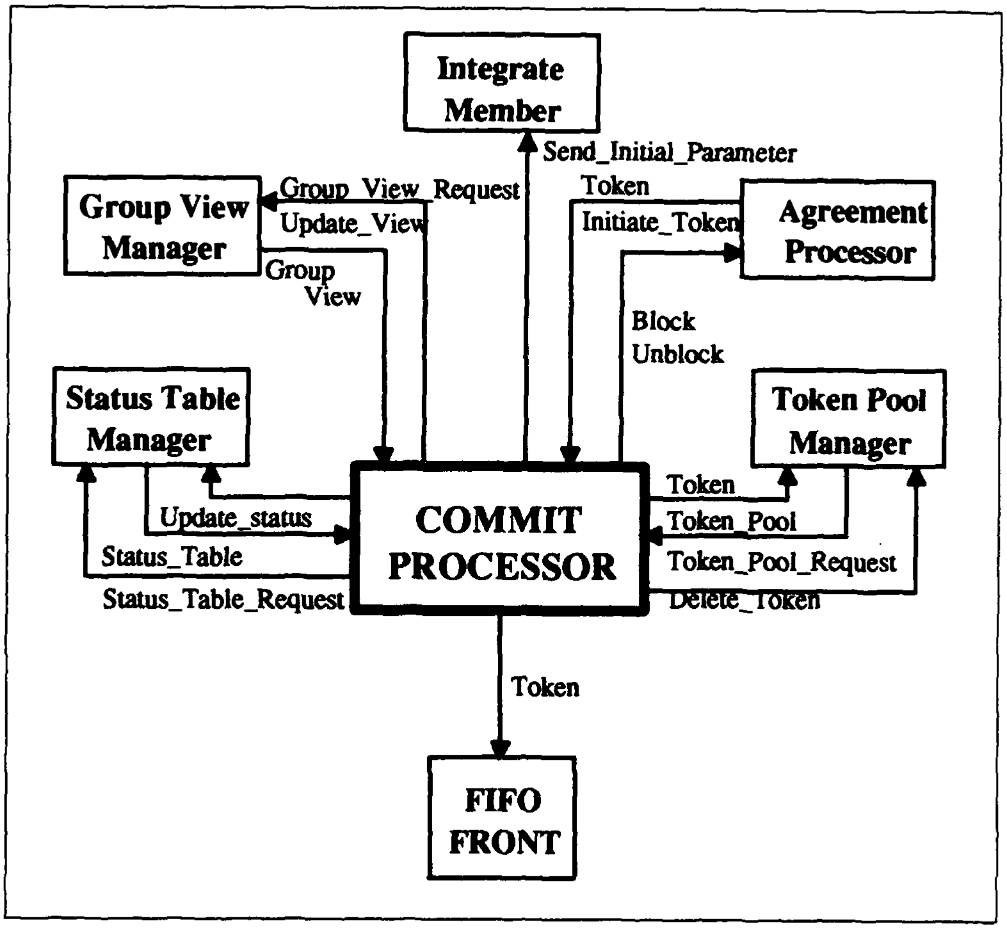

Figure A10 Commit Processor - Process Dependencies

```
CommitChange for $commit_{p_j}(p_k)$ at $p_i$
/* Depending on whether a join or departure */
add or delete $p_k$ from $GV(p_i)$
delete $p_k$ entry from $ST_{p_i}$
$vn(p_i) \leftarrow vn(p_i) + 1$
delete all commit tokens received before $agree_{p_j}(p_k)$ from $TokenPool(p_i)$
if (join committed && $joinreq_{p_j}(p_k) \in TokenPool(p_i)$ )
    delete $joinreq_{p_j}(p_k)$
end
add $commit_{p_j}(p_k)$ to $TokenPool(p_i)$
delete $agree_{p_j}(p_k)$
if (current host = $p_k$)
    determine new $p_{host}$
end
if ((join committed) && ($p_{host} = p_i$))
    send $ST_{p_i}$, $TokenPool(p_i)$, and $GV(p_i)$ to $acwnbr(p_i)$
end
send $commit_{p_j}(p_k)$ token to $cwnbr(p_i)$

end CommitChange
```

Figure A11 Actions for Committing a Change

```
ProcessCommitTkn for $commit_{p_j}(p_k)$ at $p_i$

if (initiate commit message received)
    generate commit token
    token to be processed ← generated token
else if (($p_i \neq p_j$) && (not duplicate))
    token to be processed ← received token
else
    exit
end
CommitChange
while ( $p_l \in ST_{p_i}$ with pending status & $Rank(p_l) < Rank(p_m)$, $p_m \in ST_{p_i}$)
    generate commit token
    token to be processed ← generated token
    CommitChange in rank order
end
```

Figure A12 Generate / Receive and Process a Commit Token

```
/*************************************************************************
 * Commit Processor:
 * Algorithm:   Fig. 6-7 of the gmp paper
 *
 * Written by:  Fernando Pires
 *              David Pezdirtz
 *
 * Date:        9 Dec 1993
 *************************************************************************/
#include "gmputil.c"

struct node{
  int rank;
  char*data;
  struct node*next; };

typedef struct node   node;

void Commit_Change();
void DeleteTokens();

node *order_pending();
node *make_node();

main(argc,argv)
int  argc;
char  *argv[];

{
  int       soc, newsoc, gvmfd, tpmfd, stmfd, clen, msglen, msgtype,
             Process_token;
  char      tkpreq[NODATAMSG+1+1], token[TOKENMSGLEN+1],
            gvreq[NODATAMSG+1],
             streq[NODATAMSG+1], *myaddr, *buf, *gv, *tpool, *sttable, *tkndata,
             *originator, *stmsoc, *gvmsoc, *tpmsoc, *intmbrsoc, *frontsoc,
             *tmp_token, *tmp, msg[10];
  link       *msglist;
  node       *head, *ptr;
  struct sockaddr_un   caller_addr;
```

```
printf("\tCOMMIT: start execution.\n");

/*
 * Get socket file descriptors from the command line. myaddr is  (ipaddr;front;back)
 */

if (argc != 8) {
  printf("\tUsage: Commit soc myaddr stmsoc gvmsoc tpmsoc intmbrsoc  frontsoc\n");
 printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
 exit(-1); }

else {
 soc       = atoi(argv[1]);
 myaddr    = argv[2];
 stmsoc    = argv[3];
 gvmsoc    = argv[4];
 tpmsoc    = argv[5];
 intmbrsoc = argv[6];
 frontsoc  = argv[7]; }

while (TRUE) {

 /* read message on local socket */
 clen = sizeof(caller_addr);

 if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
   printf("\tCOMMIT: accept error errno %d\n", errno);
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

 if ( (msglen = readmsg(newsoc, &buf, "#")) <0 ) {
  printf("\tCOMMIT: read error\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

 buf[msglen]=NULL;

 msgtype = in_msg_type(buf);
```

```
/* get local token pool */
tpool = GetTokenPool(tpmsoc);

/* get group view */
gv = GetGroupView(gvmsoc);

/* get status table */
sttable = GetStatusTable(stmsoc);

Process_token = FALSE;

if (msgtype == INITTOKEN){
   /* create commit token */

   /* assemble token */
   msglist = str2list(buf, "\n#");

   if ( getfromlist(msglist, &tkndata, 2 ) !=2) {
       printf("\tCOMMIT: inittoken msg parsing failed\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

   strcpy(token, tkndata);
   strcat(token, " ");
   strcat(token, myaddr);
   strcat(token, "#");

   removelist(msglist);
   Process_token = TRUE;

} /* end then msgtype = INITTOKEN */

else {

    /* process incoming token */
```

```
    /* strip off the "tokentokn" field */
   tmp_token = CALLOC( strlen(buf) + 1, char);
   strcpy(tmp_token, buf);
  msglist = str2list(tmp_token,"\n#");

  if (getfromlist(msglist, &tmp, 2) != 2) {
     printf("\tCOMMIT: token msg parsing failed\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  strcpy(token,tmp);

  removelist(msglist);
  free(tmp_token);

  /* get the originator */
  msglist = str2list(buf, "\n #");

  if ( getfromlist(msglist, &originator, 4 ) !=4) {
     printf("\tCOMMIT: token msg parsing failed\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ((strcmp(originator,myaddr) != 0) && (first_time(token, gv, sttable)))
     /* commit the change */
     Process_token = TRUE;
  else
     /* change already committed */
     Process_token = FALSE;

  removelist(msglist);

} /* end else msgtype != INITTOKEN */

free(buf);
```

```
    if (Process_token) {

      head = order_pending(tpool, gv, sttable, token, myaddr);

      while (head != NULL){

        tmp_token = CALLOC(strlen(head->data) + 1, char);
        strcpy(tmp_token, head->data);

        Commit_Change(tpool, tmp_token, myaddr, gvmsoc, tpmsoc, stmsoc, intmbrsoc,
frontsoc);

        ptr = head;
        head = ptr->next;

        free(ptr->data);
        free(ptr);
        free(tmp_token);

      } /* end while ptr */

    }  /* end if (Process_token) */

    strcpy(msg, "unblock#");
    msglen = strlen(msg);

    if ( writemsg(newsoc, msg, msglen) < msglen ) {
      printf("\tCommit: unblock agree failed\n");
     printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
     exit(-1); }

    close(newsoc);
    free(tpool);
    free(gv);
    free(sttable);

  } /* while (TRUE) */

} /* end main */
```

```
/*****************************************************************************
order_pending:  creates a linked list of the commit processes.

******* *********************************************************************/
node *order_pending(tpool, gv, sttable, token, myaddr)
char *tpool, *gv, *sttable, *token, *myaddr;

{
  link     *tplist, *tklist, *stlist, *tkn_list;
  node    *head, *point, *last, *temp, *ptr;
  int       tp_ptr, agree, pending, rank, st_ptr, done, finish, found, tpsize,
            lowest_rank;
  char     *nxt_token, *tp_subj, *tk_type, tpsizestr, *temp_tp, *temp_st,
            *tktemp, *subject, *st_subj, *status, *temp_tk,
            *new_token;

  /* insert commit token at begining of list */
  head = make_node((-1), token);

  /* disassemble token pool */
  temp_tp = CALLOC(strlen(tpool) + 1, char);
  strcpy(temp_tp,tpool);
  tplist = str2list(temp_tp, "\n=#");

  /* disassemble status table */
  temp_ st = CALLOC(strlen(sttable) + 1, char);
  strcpy(temp_st,sttable);
  stlist = str2list(temp_st, "\n= #");  /* break out elements and status */

  /* get current token subject */
  temp_tk = CALLOC(strlen(token) + 1, char);
  strcpy(temp_tk, token);
  tkn_list = str2list(temp_tk, "\n #");

  /* get commit token subj */
  if ( getfromlist(tkn_list, &subject, 2 ) != 2 ) {
    printf("\tProcCommitPending: token subject parsing failed\n");
   printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }
```

```
/* parse status table - add all members to list in rank order */
st_ptr = 3;
finish = FALSE;

while (finish != TRUE ) {

  if ( getfromlist(stlist, &st_subj, st_ptr) != st_ptr)
    finish = TRUE;
  else {
    /* get the status */
    if (getfromlist(stlist, &status, st_ptr+1) != (st_ptr + 1)){
        printf("\torder pending: status parsing failed\n");
      printf("\\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    /* check for joinreqst status ... ignore it */
    if (strcmp(status, "joinrqstd") != 0){

      /* get the corresponding agree token */
      done = FALSE;
      tp_ptr = 3;

      while (done != TRUE){

        /* check for end of token pool */
        if (getfromlist(tplist, &nxt_token, tp_ptr) != tp_ptr){
            printf("\torder pending: token not found\n");
          printf("\\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }
        else{

          /* check for proper token */
          tktemp = CALLOC(strlen(nxt_token) + 1, char);
          strcpy(tktemp, nxt_token);
          tklist = str2list(tktemp, "\n #");

          /* get next token type */
          if ( getfromlist(tklist, &tk_type, 1 ) != 1 ) {
              printf("\torder pending: token type parsing failed\n");
            printf("\\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

          /* get next token subj */
          if ( getfromlist(tklist, &tp_subj, 2 ) != 2 ) {
              printf("\torder pending: token subject parsing failed\n");
            printf("\\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

          /* if agree token */
          agree = agreetoken(nxt_token);

          if ((agree) && (strcmp(tp_subj, st_subj) == 0))
            done = TRUE;

        } /* end else */

        /* go back and look for next token in token pool */
        tp_ptr++;

        /* free storage ONLY if token not found */
        if (done == FALSE){
          removelist(tklist);
          free(tktemp);
        }

      } /* end while !done */

      /* get rank */
      if ( (rank = GetRank(gv, tp_subj)) < 0) {

        /* member not in GV ---> MUST be a join */
        rank = HIGHRANKVALUE;

      } /* end if rank < 0 */
/* find insertion point */
last = head;
point = last->next;

while ( (point != NULL) && (rank > point->rank) ) {
  last = point;
  point = point->next; }

/* create commit token */
new_token = CALLOC(TOKENMSGLEN+1, char);

if ((strcmp(status,"joinpendg") == 0) || (strcmp(status,"failpendg") == 0)){
  if (strcmp(status,"joinpendg") == 0)
    strcpy(new_token, "joincomit ");

  if (strcmp(status,"failpendg") == 0)
    strcpy(new_token, "failcomit ");

  strcat(new_token, tp_subj);
  strcat(new_token, " ");
  strcat(new_token, myaddr);
  strcat(new_token, "#");
}
else{
  strcpy(new_token, "agree");
}

/* don't insert a token for the corresponding commit token */
if (strcmp(tp_subj, subject) != 0){

  /* insert in order */
  temp = make_node(rank, new_token);
  temp->next = last->next;
  last->next = temp;
}
free(new_token);
removelist(tklist);
free(tktemp);
} /* end if status != joinrqstd */
```

```
    /* incriment pointer for next status */
    st_ptr = st_ptr + 2;

  } /* end if next status */
} /* end while !finish */

removelist(tkn_list);
removelist(tplist);
removelist(stlist);
free(temp_tk);
free(temp_tp);
free(temp_st);

/* all tokens w/ subj in status table now ordered
 * discard all past first "agree"
 *
 * NOTE: the first "agree is NOT the corresponding agree for this
 *     commit.
 */
last = head;
point = last->next;
finish = FALSE;

while (finish != TRUE){

  if (point == NULL)
    finish = TRUE;
  else{
    if (strcmp(point->data, "agree") == 0){
      /* remove the end of the list */
      last->next = NULL;
      while (point != NULL){
        ptr = point;
        point = ptr->next;
        free(ptr->data);
        free(ptr);
      }
```

```
        finish = TRUE;
      }
      else{ /* continue parse */
        last = point;
        point = last->next;
      }

    } /* end if point != NULL */

  } /* end !finish */

  return(head);

} /* end order_pending */

/*************************************************************************
make_node: creates a node
    node -> rank = rank of current
    node -> data = data
    node -> next = NULL

  returns the node ptr to the created node.
*************************************************************************/
node *make_node(rank, data)
int  rank;
char *data;

{
  node *tmp;

  tmp = CALLOC(1, node);
  tmp->rank = rank;
  tmp->data = CALLOC(strlen(data) + 1, char);
  strcpy(tmp->data,data);
  tmp->next = NULL;

  return(tmp);

} /* end make_node */
```

```
/*************************************************************************
Commit_Change:

*************************************************************************/
void Commit_Change(tpool, token, myaddr, gvmsoc, tpmsoc, stmsoc, intmbrsoc, frontsoc)
char *tpool, *token, *myaddr, *gvmsoc, *tpmsoc, *stmsoc,
       *intmbrsoc, *frontsoc;

{
  int       msglen, gvmfd, tpmfd, stmfd, intmbrfd, frontfd, joinc;
  char      *tktemp, *tktype, *subject, *originator, updtview[UPDTVIEWLEN+1],
            deltkn[DELTKNLEN+1 +2], gvreq[NODATAMSG+1], sndinip[SNDINIPLEN+1],
            updtst[UPDTSTSMSGLEN+1], *gview, *host, *ttoken, *ttpool,
            tkpreq[NODATAMSG + 1], *faux_token;

  link      *tklist;

  /* set joincommit token flag */
  joinc = FALSE;

  /* get token elements */
  tktemp = CALLOC(strlen(token) + 1, char);
  strcpy(tktemp,token);
  tklist = str2list(tktemp, "\n #");

  if ( getfromlist(tklist, &tktype, 1 ) != 1 ) {
     printf("Commit_Change: token type parsing failed\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(tklist, &subject, 2 ) != 2 ) {
     printf("Commit_Change: token subject parsing failed\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(tklist, &originator, 3 ) != 3 ) {
     printf("Commit_Change: token originator parsing failed\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
if ( (strcmp(tktype,"failcommit")==0) ) {
  /* assemble update view (delete) message */
  strcpy(updtview, "updatview\ndel ");
  strcat(updtview, subject);
  strcat(updtview, "#");

  msglen = strlen(updtview);
  gvmfd  = connectUN(gvmsoc);

  if ( writemsg(gvmfd, updtview, msglen) < msglen ) {
     printf("\tCommit_Change: update view tx failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(gvmfd);

} /* end else 'failagree' */
else {
  /* token is a joincommit */
  joinc = TRUE;

  /* assemble update view (add) message */
  strcpy(updtview, "updatview\nadd ");
  strcat(updtview, subject);
  strcat(updtview, "#");

  msglen = strlen(updtview);
  gvmfd  = connectUN(gvmsoc);

  if ( writemsg(gvmfd, updtview, msglen) < msglen ) {
     printf("\tCommit_Change: update view tx failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(gvmfd);

  /* get updated group view */
  gview = GetGroupView(gvmsoc);
```

```
/* assemble the faux joinreqst token */
faux_token = CALLOC( TOKENLEN + 1, char);
strcpy(faux_token, "joinreqst ");      /* tokentkn field not required */
strcat(faux_token, subject);
strcat(faux_token, " ");
strcat(faux_token, myaddr);            /* originator field not checked for
                                        * determination if token in pool */
strcat(faux_token, "#");

/* delete joinreqst token if I'm not host and token is in pool */
if ((GetRank(gview,myaddr) != 0) && (InTokenPool(tpool, faux_token))) {

  /* assemble delete token (joinreqst) message */
  strcpy(deltkn, "deltoken\njoinreqst ");

  strcat(deltkn, subject);
  strcat(deltkn, " ");
  strcat(deltkn, originator);
  strcat(deltkn, "#");

  msglen = strlen(deltkn);
  tpmfd  = connectUN(tpmsoc);

  if ( writemsg(tpmfd, deltkn, msglen) < msglen ) {
     printf("\tCommit_Change:delete token tx failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(tpmfd);

} /* end if GetRank && InTokenPool */

free(faux_token);

if ( GetMembWithRank(gview, &host, 0) != 0 ) (
   printf("\tCommit_Change: get host failed\n");
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$. $$$$$$$$$$\n");
  exit(-1); }
  free(gview);

} /* end else assemble update view */

/* assemble update status (delete) message */
strcpy(updtst, "updstatus\n");
strcat(updtst, subject);
strcat(updtst, " delstatus #");

msglen = strlen(updtst);
stmfd = connectUN(stmsoc);

if ( writemsg(stmfd, updtst, msglen) < msglen ) {
   printf("\tCommit_Change: update status tx failed\n");
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

close(stmfd);

/* get local token pool */
ttpool = GetTokenPool(tpmsoc);

DeleteTokens(ttpool, token, tpmsoc);

/* send token to TPM */
ttoken = CALLOC(strlen(token) + 9 + 1 + 1, char);
strcpy(ttoken,"tokentokn\n");
strcat(ttoken,token);

msglen = strlen(ttoken);
tpmfd  = connectUN(tpmsoc);

if ( writemsg(tpmfd, ttoken, msglen) < msglen ) {
   printf("\tCommit_Change: token tx failed\n");
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

close(tpmfd);
```

```
/* assemble delete agree token message */
strcpy(deltkn, "delttoken\n");

if ( strcmp(tktype, "failcomit") == 0 )
  strcat(deltkn, "failagree ");
else
  strcat(deltkn, "joinagree ");

strcat(deltkn, subject);
strcat(deltkn, " ");
strcat(deltkn, originator);
strcat(deltkn, "#");

msglen = strlen(deltkn);
tpmfd  = connectUN(tpmsoc);

if ( writemsg(tpmfd, deltkn, msglen) < msglen ) {
  printf("\tCommit_Change:delete token tx failed\n");
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

close(tpmfd);

if ((joinc) && (strcmp(host, myaddr) == 0 )) {

  /* ensure that the changes have been incorporated in the
     token pool prior to sending the initial parameters
     NOTE: async consideration                        */

  free(ttpool);
  ttpool = GetTokenPool(tpmsoc);

  /* send SendInitialParameters */
  strcpy(sndinip, "sndinipar\n");
  strcat(sndinip, subject);
  strcat(sndinip, "#");

  msglen   = strlen(sndinip);
  intmbrfd = connectUN(intmbrsoc);
```

```
    if ( writemsg(intmbrfd, sndinip, msglen) < msglen ) {
      printf("\tCommit_Change: sndinipar tx failed\n");
      printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    close(intmbrfd);

  } /* end if strcmp */

  free(host);

  /* send token to FRONT -> cwnbr */
  msglen  = strlen(ttoken);
  frontfd = connectUN(frontsoc);

  if ( writemsg(frontfd, ttoken, msglen) < msglen ) {
     printf("\tCommit_Change: token tx to front failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(frontfd);

  removelist(tklist);
  free(tktemp);
  free(ttoken);
  free(ttpool);

} /* end Commit_Change */
```

```
/*****************************************************************************
DeleteTokens:

*****************************************************************************/
void DeleteTokens(tpool, token, tpmsoc)
char *tpool, *token, *tpmsoc;

{
  int      tpsize, i, msglen, tpmfd;
  char     *tptemp, *poolsize, *tktemp, *subject, *nxttkn, *nxttktemp,
           *nxttype, *nxtsubj, *nxtorig, deltkn[DELTKNLEN+1];
  link      *tplist, *tklist, *nxttklist;

  /* disassemble token pool */
  tptemp = CALLOC(strlen(tpool) + 1, char);
  strcpy(tptemp,tpool);
  tplist = str2list(tptemp, "\n=#");

  if ( getfromlist(tplist, &poolsize, 2 ) != 2 ) {
    printf("\tProcCommitPending: token pool size parsing failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  tpsize = atoi(poolsize);

  /* get token subject */
  tktemp = CALLOC(strlen(token) + 1, char);
  strcpy(tktemp,token);
  tklist = str2list(tktemp, "\n #");

  if ( getfromlist(tklist, &subject, 2 ) != 2 ) {
     printf("\tProcCommitPending: token subject parsing failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  for (i=3; i<=tpsize + 2; i++) { /*for all tokens in tpool*/
```

```
if ( getfromlist(tplist, &nxttkn, i) != i ) {
   printf("\tProcCommitPending: token pool parsing failed for token %d\n", i);
   printf("\t\tpoolsize = %d\n", tpsize);
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

/* get next_token fields */
nxttktemp = CALLOC(strlen(nxttkn) + 1, char);
strcpy(nxttktemp, nxttkn);
nxttklist = str2list(nxttktemp, " #");

if ( getfromlist(nxttklist, &nxttype, 1 ) != 1 ){
   printf("\tProcCommitPending: next token type parsing failed\n");
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

if ( getfromlist(nxttklist, &nxtsubj, 2 ) != 2 ) {
   printf("\tProcCommitPending: next token subject parsing failed\n");
  printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

/* stop at agree(subject) */
if ((agreetoken(nxttkn)) && (strcmp(nxtsubj, subject)==0))
  break;

/* delete token */
if ( comittoken(nxttkn)){

  /* assemble delete token message */
  strcpy(deltkn, "delttoken\n");
  strcat(deltkn, nxttype);
  strcat(deltkn, " ");
  strcat(deltkn, nxtsubj);
  strcat(deltkn, " ");

  if ( getfromlist(nxttklist, &nxtorig, 3 ) != 3 ) {
     printf("\tProcCommitPending: next token originator parsing failed\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
      strcat(deltkn, nxtorig);
      strcat(deltkn, "#");

      msglen = strlen(deltkn);
      tpmfd  = connectUN(tpmsoc);

      if ( writemsg(tpmfd, deltkn, msglen) < msglen ) {
         printf("\tCommit_Change:delete token tx failed\n");
        printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      close(tpmfd);

    } /* end if strcmp(nxttype,"failcomit") */

    removelist(nxttklist);
    free(nxttktemp);

  } /* end for i */

  removelist(tplist);
  removelist(tklist);
  free(tptemp);
  free(tktemp);

} /* end DeleteTokens */
```

INTEGRATE MEMBER PROCESS

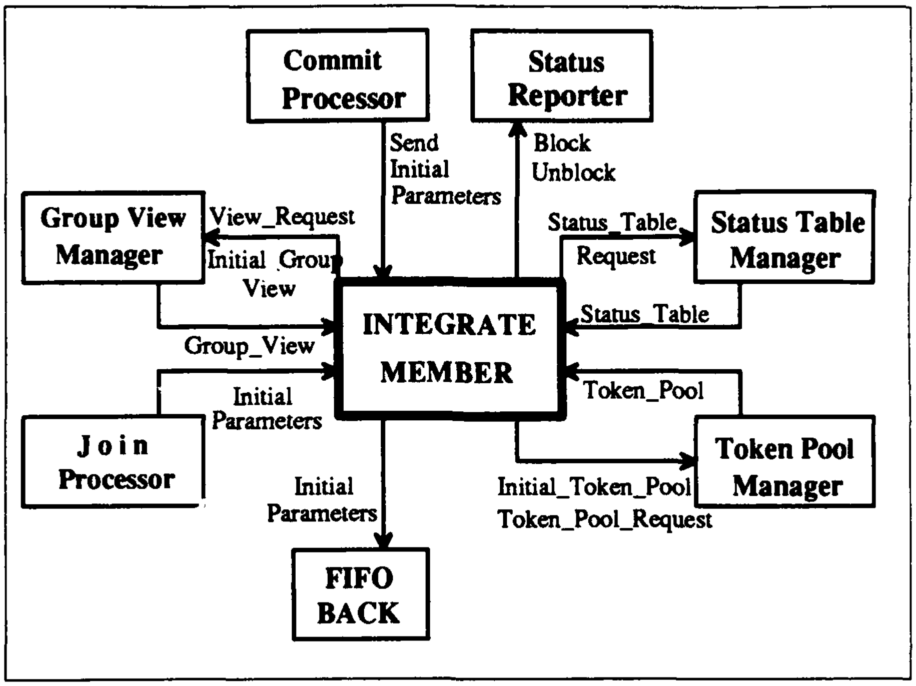

Figure A13 Integrate Member - Process Dependencies

```
IntegrateMember
if (initial parameters)
    send blocking message to status reporter
    send $GV$ to group view manager
    send unblocking message to status reporter
    send $ST$ to status table manager
    send TokenPool to token pool manager
else
    get $GV_{P_i}$ from group view manager
    get $ST_{P_i}$ from status table manger
    get TokenPool($p_i$) from token pool manager
    assemble initparam message
    send initparam message to new member
end
```

Figure A14 Integrate Member Process Specification

```
/***************************************************************************
 * Integrate Member (INTMBR)
 *
 * Version: 19 May 1993
 * Author: David Pezdirtz
 *
 * DESCRIPTION:
 *   Waits for a connection and then reads the initial parameter msg.
 *    Sends the init sub-msg to the 3 data base managers.  On initial param
 *    request, compiles the init_params and returns the init_param msg.
 *
 * USAGE: intmbr soc my_addr gvmsoc stmsoc tpmsoc backsoc
 *
 ***************************************************************************/


/***************************************************************************
  Declarations
***************************************************************************/
#include "gmp.h"
#include "msgutil.c"
#include "socutil.c"

typedef char *buffer;

void demux();
void mux();

/***************************************************************************
Main:

***************************************************************************/
main(argc,argv)
int      argc;
char     *argv[];

{
  int                        soc, newsoc, clen, msglen;
  char                       *gvmsoc, *stmsoc, *tpmsoc, *backsoc;
char                    *my_addr, *target;
int                     action;
struct sockaddr_un      caller_addr;
char                    *buf;
link                    *msgl1, *msgl2;

if ( argc == 7 ) {
  soc    = atoi(argv[1]);
  my_addr = argv[2];
  gvmsoc  = argv[3];
  stmsoc  = argv[4];
  tpmsoc  = argv[5];
  backsoc = argv[6]; }

else {
  printf("INTMBR usage error: intmbr soc my_addr gvmsoc stmsoc tpmsoc backsoc\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

while( TRUE ) {
  clen = sizeof(caller_addr);

  /* Accept connection request from client */
  clen = sizeof(caller_addr);

  if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0) {
    printf("\tINTMBR: accept error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* Read message */
  if ( (msglen = readmsg(newsoc, &buf, "#")) < 0 ) {
    printf("\tINTMBR: read error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  buf[msglen] = NULL;
```

```
    /* close the input socket */
    close(newsoc);

    /* Determines which message was received */
    action = in_msg_type(buf);

    /* set up the top level list */
    msgl1 = str2list(buf,"\n@#");

    switch (action) {
      case INITPARAM:
        demux(msgl1, gvmsoc, stmsoc, tpmsoc);
        break;


      case SNDINIPAR:
        if (getfromlist(msgl1, &target, 2) != 2) {
          printf("INTMBR error: get target\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

        mux(backsoc, gvmsoc, stmsoc, tpmsoc, target, my_addr);
        break;

      default:
        printf("INTMBR error: invalid msg type\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1);

    } /* end switch */

    /* Dispose of msg list */
    removelist(msgl1);
    free(buf);

  } /* end while */

} /* end main */
```

```
/***************************************************************************
demux: break apart the initial parameters message and send the sub-msgs
   to the individual database managers.

** ***********************************************************************/
v  l demux(msgl1, gvmsoc, stmsoc, tpmsoc)
li.x   *msgl1;
char   *gvmsoc, *stmsoc, *tpmsoc;

{
  int      gvmfd, stmfd, tpmfd, msglen;
  link     *msgl2;
  char     *msg, *temp_data;
  char     *header;

  /* get the group view */
  if (getfromlist(msgl1, &temp_data, 3) == 0) {
    printf("INTMBR: getfromlist invalid format (grpview)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* convert group view to list as only member */
  msgl2 = str2list(temp_data,"#");

  /* generate the initial group view message */
  header = CALLOC(9 + 2, char);
  strcpy(header,"initgview");

  msg = list2str(msgl2, header,"\n","#");
  msglen = strlen(msg);

  /* send the initial group view to gvm */
  gvmfd = connectUN(gvmsoc);

  if (writemsg(gvmfd, msg, msglen) < msglen) {
    printf("\tINTMBR: initgview to gvm failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
close(gvmfd);
removelist(msgl2);
free(msg);

/* get the status table */
if (getfromlist(msgl1, &temp_data, 4) == 0) {
  printf("INTMBR: getfromlist invalid format (status table)\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

/* convert status table to list as only member */
header = CALLOC(9 + 2, char);
msgl2 = str2list(temp_data,"#");

/* generate the initial status table message */
strcpy(header,"inittable");

msg = list2str(msgl2, header,"\n","#");
msglen = strlen(msg);

/* send the initial status table to stm */
stmfd = connectUN(stmsoc);

if (writemsg(stmfd, msg, msglen) < msglen) {
  printf("\tINTMBR: inittable to stm failed.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

close(stmfd);

removelist(msgl2);
free(msg);

/* get the initial token pool */
if (getfromlist(msgl1, &temp_data, 5) == 0) {
  printf("INTMBR: getfromlist invalid format (token pool)\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }
```

```
  /* convert token pool to list as only member */
  msgl2 = str2list(temp_data,"#");

  /* generate the initial token pool message */
  header = CALLOC(9 + 2, char);
  strcpy(header,"inittpool");

  msg = list2str(msgl2, header,"\n","#");
  msglen = strlen(msg);

  /* send the initial token pool to tpm */
  tpmfd = connectUN(tpmsoc);

  if (writemsg(tpmfd, msg, msglen) < msglen) {
    printf("\tINTMBR: inittpool to tpm failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(tpmfd);

  removelist(msgl2);
  free(msg);

} /* end demux */
```

```
/****************************************************************************
mux: assemble the initial parameter message from the existing
    data base managers and send the msg to the back processor.

****************************************************************************/
void mux(backsoc, gvmsoc, stmsoc, tpmsoc, target, my_addr)
char *backsoc, *gvmsoc, *stmsoc, *tpmsoc, *target, *my_addr;

{
  int       backfd, gvmfd, stmfd, tpmfd;
  char     msg[NODATAMSG + 1 + 10], *message;
  char     *gvmsg, *stmsg, *tpmsg;
  char     *gv, *st, *tp;
  int      gvmsglen, stmsglen, tpmsglen;
  int      message_len;
  int      msglen;
  link     *gvmsgl, *stmsgl, *tpmsgl;

  strcpy(msg,"viewreqst#");
  msglen = strlen(msg);

  gvmfd = connectUN(gvmsoc);

  if (writemsg(gvmfd, msg, msglen) < msglen) {
    printf("\tINTMBR: viewreqst to gvm failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* read the group view message */
  if ( (gvmsglen = readmsg(gvmfd, &gvmsg, "#")) < 0) {
    printf("Integrate Mbr unix: read error - gvm\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  gvmsg[gvmsglen] = NULL;
  close(gvmfd);

  strcpy(msg,"statreqst#");
  msglen = strlen(msg);

  stmfd = connectUN(stmsoc);

  if (writemsg(stmfd, msg, msglen) < msglen) {
    printf("\tINTMBR: statreqst to stm failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* read the status table message */
  if ( (stmsglen = readmsg(stmfd, &stmsg, "#")) < 0) {
    printf("Integrate Mbr unix: read error - stm\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  stmsg[stmsglen] = NULL;
  close(stmfd);

  strcpy(msg,"tokpreqst#");
  msglen = strlen(msg);

  tpmfd = connectUN(tpmsoc);

  if (writemsg(tpmfd, msg, msglen) < msglen) {
    printf("\tINTMBR: tokpreqst to tpm failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* read the token pool message */
  if ( (tpmsglen = readmsg(tpmfd, &tpmsg, "#")) < 0) {
    printf("Integrate Mbr unix: read error - tpm\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  tpmsg[tpmsglen] = NULL;
  close(tpmfd);

  /* convert the data base messages into the proper format */
  gvmsgl = str2list(gvmsg,"\n#");
  stmsgl = str2list(stmsg,"\n#");
  tpmsgl = str2list(tpmsg,"\n#");

  if ( getfromlist(gvmsgl, &gv, 2) == 0) {
    printf("Integrate Mbr error: getfromlist gv\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(stmsgl, &st, 2) == 0) {
    printf("Integrate Mbr error: getfromlist st\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( getfromlist(tpmsgl, &tp, 2) == 0) {
    printf("Integrate Mbr error: getfromlist tp\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* reserve temp storage for outgoing message */
  message = CALLOC((9 + 2*MAXLMTSIZE + strlen(gv) + strlen(st) + strlen(tp) + 5 + 11),
char);

  /* assemble the message */
  strcpy(message,"initparam\n");
  strcat(message,target);
  strcat(message," ");
  strcat(message,my_addr);
  strcat(message,"@");
  strcat(message,gv);
  strcat(message,"@");
  strcat(message,st);
  strcat(message,"@");
  strcat(message,tp);
  strcat(message,"#");

  message_len = strlen(message);

  /* write the message */
  backfd = connectUN(backsoc);
```

```
  if (writemsg(backfd, message, message_len) < message_len) {
    printf("Integrate Mbr unix: writemsg --- still no go\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(backfd);

  /* clean up memory allocated */
  removelist(gvmsgl);
  removelist(stmsgl);
  removelist(tpmsgl);

  free(message);
  free(gvmsg);
  free(stmsg);
  free(tpmsg);

} /* end mux */
```

JOIN PROCESSOR

```
InitiateJoin for a join request message/token for $p_{new}$ at $p_i$

while (true)
  if ($p_{new} \notin ST_{p_i}, GV_{p_i}$)
    receive join request message or token for $p_{new}$
  end
  if ($p_i = p_{host}$)
    send initiate agreement message to AgreeProcessor for $p_{new}$
    block until AgreeProcessor acknowledges end of processing
  else
    $ST_{p_i}(p_{new}) \leftarrow JoinRequested$
    if (join request message) /* $p_{new}$ locates $p_i$ and sends its join request */
      generate $joinreq_{p_i}(P_{new})$ token
    end
    add $joinreq_{p_i}(p_{new})$ to $TokenPool(p_i)$
    send joinreq token to $cwnbr(p_i)$
  end
end

end InitiateJoin
```

Figure A15 Processing of a Join Request Message / Token

```
/******************************************************************************
 * JOINPROCESSOR:
 * Algorithm:   Fig. 5 of the gmp paper
 * Description: - Uses one Unix domain socket, self address, and socket
 *          names for the agreement processor, group view manager, status
 *          table manager, token pool manager, commit processor, and
 *          integrate member.  Also receives front and back.
 *
 *          - A group name has to be supplied as a command line argument.
 *
 *          - 1 (local site) or more ip addresses may be supplied.
 *          Each is searched sequentially until one
 *          with /tmp/groupname member instance is found. The file
 *          /tmp/groupname is read.
 *          Starting with the first entry, a join request is sent to each
 *          until the group is joined or all entries are searched.
 *          If a member of the target group is not found, an attempt to
 *          the file on the next site from the command line argument
 *          read is made until all the sites are treated or the group
 *          is joined.  If all sites have been searched without a join,
 *          the group is registered locally.
 *
 *          - If join succeeds at any point, initial parameters are
 *          sent to integrate member and an infinite while loop is
 *          entered to receive and process the next join request/token.
 *
 * Written by:  Shridhar Shukla
 *              David Pezdirtz
 *
 * Date:    15 Nov 1993
 ******************************************************************************/
#include "gmputil.c"

int CopyGVFile();
int IsGroup();
int InStatusTable();
int CountDown();
int GetCountForJoin();
void SendTkn2Agr();
```

```
main(argc,argv)
int     argc;
char    *argv[];

{
  int   soc, newsoc, clen, msglen, msgtype, nsites, stmsglen,
        gvmfd, stmfd, tpmfd, intfd, backfd, frontfd, initgvlen,
        i, foundmember, groupsize, c, j, on, countval;
  char  *myaddr, *stmsoc, *gvmsoc, *tpmsoc, *agrsoc, *tinitpbuf,
        *intsoc, *backsoc, *frontsoc, *groupname, *sites, *tmp_buf,
        *initpbuf, *buf, *stbuf, *originator, request[NODATAMSG+1],
        targetmember[MAXLMTSIZE], joinrequest[QUERYLEN + 1],
        *initpmsgtype, *sitename, initgv[INITGVMSGLEN + 1],
       *token, updtstsmsg[UPDTSTSMSGLEN + 1];

  struct  sockaddr_un       caller_addr;
  link    *msglist, *sitelist, *msg;
  FILE    *fileptr;

/******************************************************************************
  * Get socket file descriptors from command line argument.
  * myaddr is  (ipaddr;front;back)
  ******************************************************************************/

  if (argc != 12) {
    printf("Usage: JoinProcessor soc myaddr stmsoc gvmsoc tpmsoc agrsoc intsoc backsoc
frontsoc groupname siteaddrsSeparatedBy=\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  else {
    soc     = atoi(argv[1]);
    myaddr  = argv[2];
    stmsoc  = argv[3];
    gvmsoc  = argv[4];
    tpmsoc          = argv[5];
    agrsoc  = argv[6];
    intsoc  = argv[7];
    backsoc  = argv[8];
  frontsoc  = argv[9];
  groupname = argv[10];
  sites     = argv[11]; }

sitelist = str2list(sites, "=");

if ( (nsites = listsize(sitelist)) == 0 ) { /* i must be >= 1 */
  printf("\tJOINPROCESSOR: site list parsing failed or empty list.\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

i = 0;
foundmember = FALSE;

while ( (foundmember == FALSF, && (++i <= nsites) ) {

  if ( getfromlist(sitelist, &sitename, i) == 0 ) {
    printf("\tJOINPROCESSOR: site list parsing failed for site %d.\n", i);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if (CopyGVFile(sitename, myaddr, groupname) == 0) {
    printf("\tJOINPROCESSOR: group view file not found at %s.\n", sitename); }

  else { /* file found and copied, if found assumed to be in the correct format */
    fileptr = fopen(groupname, "r");
    fscanf(fileptr, "%d", &groupsize);
    fscanf(fileptr, "%d", &groupsize); /* overwrite */

    c = fgetc(fileptr); /* to get rid of the \n after groupsize */

    while ( (foundmember == FALSE) && (groupsize-- > 0) ) {

          /* get the next entry as the request target */
          j = 0;

          while ( ((c = fgetc(fileptr)) != EOF) && ( c != '\n'))
            targetmember[j++] = c;
```

```
          targetmember[j] = NULL;

          /* assemble a request */
          strcpy(joinrequest, "joinreqst\n");
          strcat(joinrequest, targetmember);
          strcat(joinrequest, " ");
          strcat(joinrequest, myaddr);
          strcat(joinrequest, "#");

          /* send a join request */
          msglen = strlen(joinrequest);
          backfd = connectUN(backsoc);

          if ( writemsg(backfd, joinrequest, msglen) < msglen) {
            printf("\tJOINPROCESSOR: join request to back failed.\n");
            printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }

          close(backfd);

          /* Make soc nonblocking to enble count down while waiting */
          on = NONBLOCKING;
          ioctl(soc, FIONBIO, (char *) &on);
          countval = GetCountForJoin();

          while ( (foundmember == FALSE) && (CountDown(&countval,
COUNT_DOWN_STEP) != 0) ) {
            clen = sizeof(caller_addr);

            if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) == -1) {

                  if ( (errno == EWOULDBLOCK) || (errno == EINPROGRESS) ) {
                      ; /* no connections present */ }
                  else {  printf("\tJOINPROCESSOR: initparam accept error \n");
                      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                      exit(-1); }

            } /* end if newsoc */
            else { /* a connection is present */
```

```
                if ( (msglen=readmsg(newsoc,&initpbuf,"#")) < 0 ) {
                  printf("\tJOINPROCESSOR: initparam PORT read error.\n");
                  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                  exit(-1); }

                initpbuf[msglen] = NULL;
                tinitpbuf = CALLOC(strlen(initpbuf) + 1, char);
                strcpy(tinitpbuf,initpbuf);

                /* msg was read - check its type to be initparam */
                msglist = str2list(tinitpbuf, "\n#");

                if ( getfromlist(msglist, &initpmsgtype, 1) == 0) {
                  printf("\tJOINPROCESSOR: msgtype parsing failed.\n");
                  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                  exit(-1); }

                free(tinitpbuf);

                if ( strcmp(initpmsgtype, "initparam") != 0) {
                  printf("\tJOINPROCESSOR: initparam expected something else
rcvd.\n");

                  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                  exit(-1); }
                else foundmember = TRUE;

                /* Make soc blocking as before */
                on = BLOCKING;
                ioctl(soc, FIONBIO, (char *) &on);

           close(newsoc);

         } /* end else newsoc = accept */
       } /* end while foundmember is false & CountDown*/
     } /* end while foundmember is false & groupsize */
   } /* end else CopyGVFile */
 } /* end while foundmember is false & i++ */
```

```
if (foundmember == TRUE) { /* send initial parameters to integrate member */

  /* no changes required to the initparam msg rcvd from front earlier */
  msglen = strlen(initpbuf);
  intfd  = connectUN(intsoc);

  if ( writemsg(intfd, initpbuf, msglen) < msglen) {
          printf("\tJOINPROCESSOR: initparam msg send to integrate member failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

  close(intfd);
  removelist(msglist);
  free(initpbuf);
  free(tinitpbuf);

} /* end if foundmember = true */
else { /* send initial group view msg to GVM */

  strcpy(initgv, "initgview\n0=1=");
  strcat(initgv, myaddr);
  strcat(initgv, "#");
  initgvlen = strlen(initgv);
  gvmfd = connectUN(gvmsoc);

  if ( writemsg(gvmfd, initgv, initgvlen) < initgvlen) {
          printf("\tJOINPROCESSOR: initial group view  to gvm failed.\n");
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

  close(gvmfd);

 } /* end else foundmember */

 removelist(sitelist);
```

```
/*****************************************************************************
  * Element has become a member of the name group. Now, simply wait for
  * join requests or tokens to arrive and process them.

*****************************************************************************/
 while (TRUE) {

  /* read join request from front or join token from agreement process */
  clen = sizeof(caller_addr);

  if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0 ){
          printf("\tJOINPROCESSOR: unix accept error for join req / token.\n");
          printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

  if ( (msglen=readmsg(newsoc,&buf,"#")) < 0 ) {
          printf("\tJOINPROCESSOR: unix read error for join request or token.\n");
          printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

  buf[msglen] = NULL;
  close(newsoc);

  /* only tokens rcvd here are join tokens or joinrequests */
  msgtype = in_msg_type(buf);

  if ( (msgtype != JOINREQST) && (msgtype != TOKENTOKN) ) {
          printf("\tJOINPROCESSOR: join req or token expected. rcvd => I%sI\n", buf);
   printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

  /* get the originator of the request or subject of token
   * (3rd field in both)
   */
  tmp_buf = CALLOC(strlen(buf) + 1, char);
  strcpy(tmp_buf, buf);

  msg = str2list(tmp_buf, "\n #");

 if ( getfromlist(msg, &originator, 3) == 0 ) {
   printf("\tJOINPROCESSOR: group join request/token parsing failed.\n");
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

 /* if I'm the host -> initiate joinagree token */
 if ( InGroup(gvmsoc, myaddr) == 0 )
   SendTkn2Agr(agrsoc, "joinagree", originator);
 else {
  /* add joinreqst status to status table */
  strcpy(updtstsmsg, "updstatus\n");
  strcat(updtstsmsg, originator);
  strcat(updtstsmsg, " ");
  strcat(updtstsmsg, "joinpendg");
  strcat(updtstsmsg, "#");

  /* send update status message to status table manager */
  msglen = strlen(updtstsmsg);
  stmfd  = connectUN(stmsoc);

  if ( writemsg(stmfd, updtstsmsg, msglen) < msglen) {
    printf("\tExecuteAgreement: status update to stm failed.\n");
    printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  close(stmfd);

  /* if joinreq msg -> generate joinreqst token */
  if (msgtype == JOINREQST){
    token = CALLOC(TOKENMSGLEN + 1, char);
    strcpy(token, "tokentokn\njoinreqst ");
    strcat(token, originator);
    strcat(token, " ");
    strcat(token, myaddr);
    strcat(token, "#"); }
  else {
    token = CALLOC( strlen(buf) + 1, char);
    strcpy(token, buf);  }
```

```
      /* add joinreqst token to token pool */
      msglen = strlen(token);
      tpmfd  = connectUN(tpmsoc);

      if ( writemsg(tpmfd, token, msglen) < msglen) {
        printf("\tProcessToken: tkn send to join process failed.\n");
        printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      close(tpmfd);

      /* send joinreqst token to cwnbr */
      msglen  = strlen(token);
      frontfd = connectUN(frontsoc);

      if ( writemsg(frontfd, token, msglen) < msglen) {
        printf("\tAgreement: token send to  front failed.\n");
        printf("\t\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      close(frontfd);

      free(token);

    } /* end else InGroup */

    removelist(msg);
    free(buf);
    free(tmp_buf);

  } /* end while true */

} /* end JOINPROCESSOR */
```

```
/******************************************************************************
CopyGVFile: Returns a 0 if file not found at the site ip address sitename.
        Else returns a 1 after copying the file in the local directory
        with name as groupname.
******************************************************************************/
int CopyGVFile(sitename, myaddr, groupname)
char *sitename, *myaddr, *groupname;

{
  int       copied;
  u_long    inaddr;
  char      command[512], *ip_address, localaddr[MAXLMTSIZE+1], *myname;
  FILE      *fileptr;
  struct    hostent hentry, *hptr;
  link      *myaddrlist;

  hptr = &hentry;

  if ( (inaddr = inet_addr(sitename)) != INADDR_NONE) {
    bcopy((char *) &inaddr, (char *)hptr->h_addr, sizeof(inaddr)); }
  else {
    if (hptr=gethostbyname(sitename))
      printf("\tCopyGVFile: Success, found %s , also known as %s\n",
         hptr->h_name,hptr->h_aliases[0]);
    else
      printf("\tCopyGVFile: Sorry host %s not found\n",sitename);

  } /* end else inaddr */

  ip_address = (char*) inet_ntoa( *((struct in_addr*)( hptr->h_addr)));

  strcpy(localaddr, myaddr);
  myaddrlist = str2list(localaddr, ";");

  if ( getfromlist(myaddrlist, &myname, 1) != 1) {
    printf("\tCopyGVFile: my addr parsing failed.\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
  if ( strcmp(myname, ip_address) != 0) { /* need to attempt copying */
    sprintf(command, "rsh %s cat %s > %s", sitename, groupname, groupname);

    if ( system(command) != 0 ) {
            printf("\tCopyGVFile: Could not rsh at site %s.\n", sitename);
            copied = 0; }
    else {
            sprintf(command, "chmod 777 %s", groupname);

            if ( system(command) != 0) {
                 printf("\tCopyGVFile: Could not change mode.\n");
                 printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
                 exit(-1); }

    } /* end else system */

  } /* end if strcmp */

  fileptr = fopen(groupname, "r");

  if (fileptr) {

    if (fgetc(fileptr) != EOF)
            copied = 1;
    else copied = 0;

    fclose(fileptr); }

  else copied = 0;

  removelist(myaddrlist);

  return(copied);

} /* end CopyGVFile */
```

```
/********************************************************************************
GetCountForJoin: Determines the wait before the join request is repeated.
            (value returned times the COUNT_DOWN_STEP seconds)
********************************************************************************/
int GetCountForJoin()
{
  int      next;

  next = RESENDREQST;
  return(next);

} /* end GetCountForJoin */
```

DATABASE MANAGERS

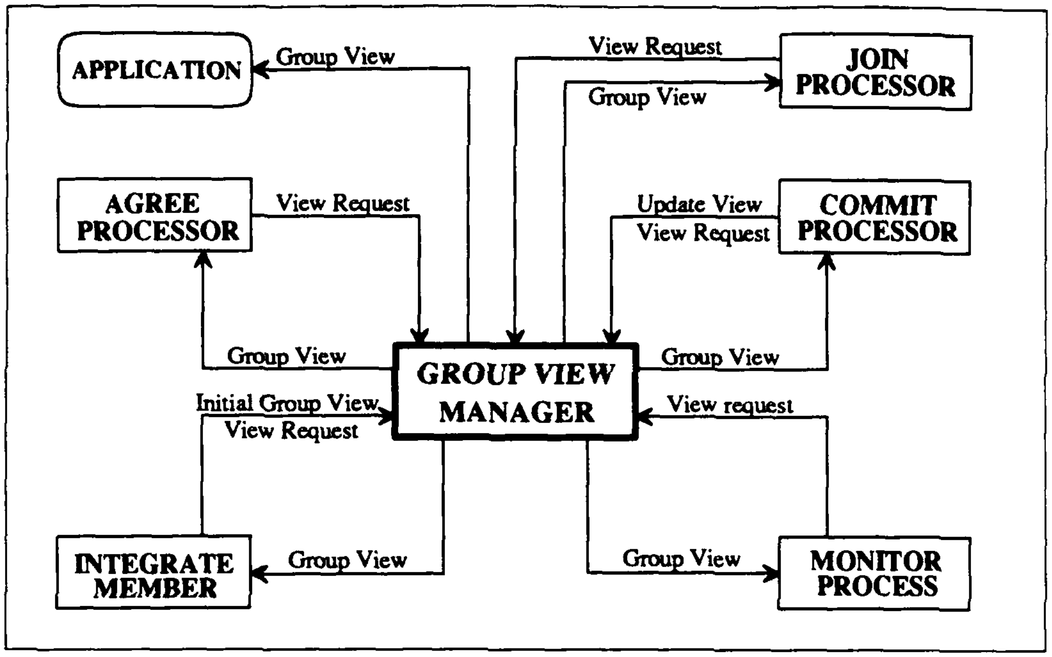

Figure A16 Group View Manager - Process Dependencies

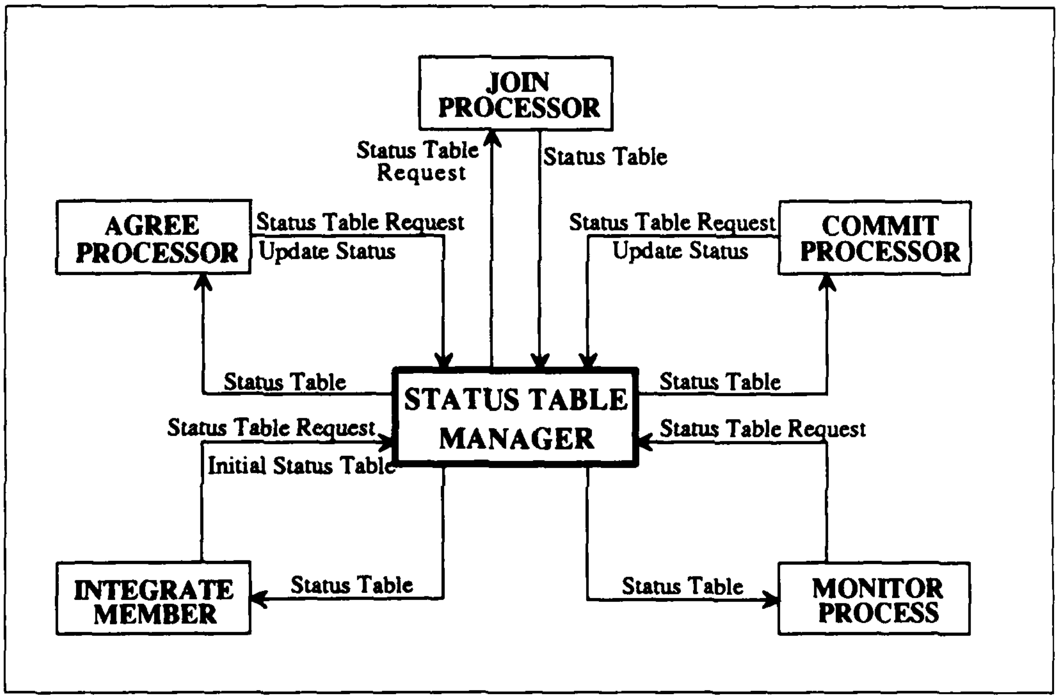

Figure A17 Status Table Manager - Process Dependencies

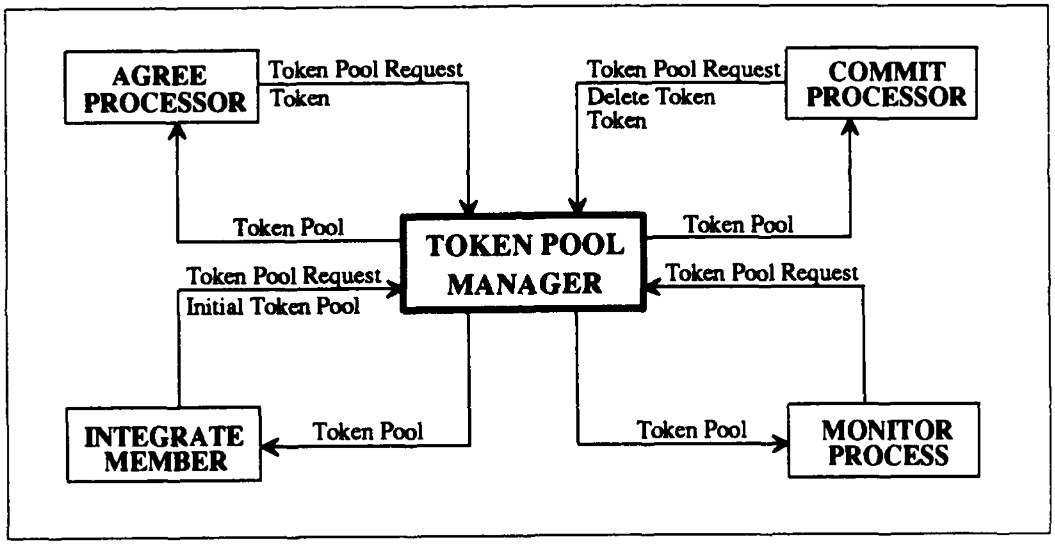

Figure A18 Token Pool Manager - Process Dependencies

```
/*************************************************************************
 * Group View Manager (GVM)
 *
 * Version: 06 APR 1993
 * Author: David Pezdirtz
 *
 * DESCRIPTION:
 *     Waits for a connection and then reads a message.  Depending on the
 * message type it executes one particular action.  Dumps the group view
 * to a group file and sends the group view to the application socket on
 * every update to the group view.
 *
 * USAGE: gvm soc initial_element appsoc group_name
 *
 *************************************************************************/

/*************************************************************************
  Declarations
 *************************************************************************/


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

#define ADD      1                  /* possible update actions */
#define DEL      2

typedef char *buffer;

void     create_msg();
void     process_update();
void     process_request();
void     initialize();
void     add();
void     del();
void     remove_view();
void     save_grp_view();

void show_grp_view();
```

```
/*************************************************************************
Main :

 *************************************************************************/
main(argc,argv)
int       argc;
char      *argv[];

{
  int                        soc,newsoc,clen,msglen, appfd;
  int                        action;
  struct sockaddr_un         caller_addr;
  char                       *msgtype, *buf, *tmp, *grp_name, *appsoc;
  link                       *init_element;
  link                       *head = NULL;
  link                       *msgl1;
  int                        view_nmbr = 0;
  int                        view_size = 1;

  if ( argc == 5 ){
    soc = atoi(argv[1]);

    tmp = CALLOC(HEADERSIZE + 4 + strlen(argv[2]), char);
    strcpy(tmp,"initheadr\n0=1=");
    strcat(tmp,argv[2]);

    appsoc = argv[3];
    grp_name = argv[4]; }
  else {
    printf("GVM usage error: gvm soc initial_element app_soc grp_name\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* initialize the group view for a single member */
  init_element = str2list(tmp,"\n");
  initialize(&head, init_element, &view_nmbr, &view_size);

  removelist(init_element);
  free(tmp);
while( TRUE ){
  clen = sizeof(caller_addr);

  /* Accept connection request from client */
  clen = sizeof(caller_addr);
  if ( (newsoc = accept(soc,(struct sockaddr*) &caller_addr, &clen)) < 0) {
    printf("GVM: accept error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* Read message */
  if ( (msglen = readmsg(newsoc, &buf, "#")) < 0 ) {
    printf("GVM: read error\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  buf[msglen] = NULL;

  /* Determines which message was received */
  action = in_msg_type(buf);

  /* set up the top level list */
  msgl1 = str2list(buf,"\n#");

  switch (action) {
    case VIEWREQST:
      process_request(newsoc, head, view_nmbr, view_size);
      break;

    case UPDATVIEW:
      process_update(&head, msgl1, &view_nmbr, &view_size);
      save_grp_view(grp_name, head, view_nmbr, view_size);
      appfd = connectUN(appsoc);
      process_request(appfd, head, view_nmbr, view_size);
      close(appsoc);
      break;

    case INITGVIEW:
      initialize(&head, msgl1, &view_nmbr, &view_size);
      save_grp_view(grp_name, head, view_nmbr, view_size);
      appfd = connectUN(appsoc);
      process_request(appfd, head, view_nmbr, view_size);
      close(appsoc);
      break;

    default:
      printf("GVM error: invalid msg type %s\n",msgtype);
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1);

  } /* end switch */

  /* Dispose of msg list */
  removelist(msgl1);
  free(buf);

  close(newsoc);

} /* end while */

} /* end main */
```

```
/****************************************************************************
process_update : process the update group view message

usage: process_update(link *head, char *msgl1)
       head - ptr to the qroup view linked list

****************************************************************************/
void process_update(head,msgl1,view_nmbr,view_size)
link   **head;
link   *msgl1;
int    *view_nmbr,*view_size;
{
  int      action;
  char     *request, *element, *trqst;
  link     *msgl2;

  action = 0; /* set for default */

  /* get the action */
  if ( (getfromlist(msgl1, &request, 2)) == 0 ) {
    printf("GVM process_update error: getfromlist(1)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  trqst = CALLOC(strlen(request) + 1, char);
  strcpy(trqst, request);

  msgl2 = str2list(trqst," #");

  if ( (getfromlist(msgl2, &request, 1)) == 0 ) {
    printf("GVM process_update error: getfromlist(2)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( strcmp(request,"add") == 0)
    action = ADD;

  if ( strcmp(request,"del") == 0)
    action = DEL;
```

```
  if ( (getfromlist(msgl2, &element, 2)) == 0) {
    printf("GVM process_update error: getfromlist element failed\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  switch (action) {
    case ADD :
      add(head,element);
      *view_size = ++*view_size;
      break;

    case DEL :
      del(head,element);
      *view_size = --*view_size;

      if (*view_size < 1) {
        printf("GVM error: deleted last group member\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      break;

    default :
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1);

  } /* end switch */

  *view_nmbr = ++*view_nmbr;

  removelist(msgl2);
  free(trqst);

} /* end process_update */
```

```
/************************************************************************
process_request : assemble the list into a string and write the message to
            the socket

output msg format:
"groupview"|\n|view #|=|view size|=|element|=|element| ... |#|

************************************************************************/
void process_request(soc, view, nmbr, size)
int    soc;
link   *view;
int    nmbr, size;

{
  char temp[MAXNUM + 1];
  char *header, *msg;
  int  msglen;

  header = CALLOC((9 + 2*MAXNUM + 3 + 1), char);

  msg = CALLOC((9 +2*MAXNUM + 3 + size*(MAXLMTSIZE + 1) + 1), char);

  /* create the msg header */
  strcpy(header,"groupview");
  strcat(header,"\n");

  sprintf(temp,"%i", nmbr);
  strcat(header,temp);

  strcat(header,"=");
  sprintf(temp,"%i", size);
  strcat(header,temp);

  /* create the message */
  msg = list2str(view, header, "=", "=");

  strcat(msg,"#");

  msglen = strlen(msg);

  if ( (writemsg(soc,msg,msglen)) != msglen) {
    printf("GVM process_request error: writemsg\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  free(msg);
  free(header);

} /* end process_request */

/************************************************************************
add : add a node to the end of the current group view.

************************************************************************/
void add(head, element)
link **head;
char *element;
{
  link *tmp, *ptr;

  tmp = CALLOC(1,link);
  tmp->data = CALLOC(strlen(element) + 1, char);
  strcpy(tmp->data,element);
  tmp->next = NULL;

  if (*head == NULL)          /* check for empty list */
    *head = tmp;
  else {                      /* parse to end of list */
    ptr = *head;

    while ( ptr->next != NULL) {
        ptr = ptr->next; }

    ptr->next = tmp;

  } /* end else head */

} /* end add */
```

```
/****************************************************************************
del : search for the specified element and delete the node from the list.
    Exit on error conditions:

                    1. empty list
                    2. element not found in list

****************************************************************************/
void del(head,element)
link **head;
char *element;

{
  link *ptr,*last;
  int found;

  found = FALSE;

  ptr = *head;
  last = NULL;

  if (ptr == NULL){
    printf("GVM del element: empty list\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }


  while ( (found == FALSE) && (ptr != NULL)) {

    if ( strcmp(ptr->data, element) == 0)
      found = TRUE;
    else {
      last = ptr;
      ptr = ptr->next; }

  } /* end while found */
```

```
  /* ERROR: requested delete not in list */
  if (found == FALSE) {
    printf("GVM del element: !%s! not in list\n", element);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* check for delete of 1st element */
  if (last == NULL)
    *head = ptr->next;                /* if so, reset the head ptr */
  else
    last->next = ptr->next;           /* otherwise remove node from list */

  /* free up the storage used */
  free(ptr->data);
  free(ptr);

} /* end del */
```

```
/****************************************************************************
initialize: initialize the group view.  Delete and removes existing list.

****************************************************************************/
void initialize(head, list, nmbr, size)
link  **head, *list;
int    *nmbr, *size;

{
  link    *msgl;
  int      next_element;
  char    *data, *tmp_data;

  if (*head != NULL) {
    remove_view(*head);
    *head = NULL; }

  if ( (getfromlist(list, &data, 2)) == 0) {
    printf("GVM initialize error: invalid format a\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  tmp_data = CALLOC( strlen(data) + 1, char);
  strcpy(tmp_data, data);

  msgl = str2list(tmp_data,"=#");

  if ( (getfromlist(msgl, &data, 1)) == 0) {
    printf("GVM initialize error: invalid format b\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  *nmbr = atoi(data);

  if ( (getfromlist(msgl, &data, 2)) == 0) {
    printf("GVM initialize error: invalid format c\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  *size = atoi(data);

  next_element = 3;

  while ( (getfromlist(msgl, &data, next_element)) > 0) {
    add(head,data);
    next_element = ++next_element; }

  if (next_element < 4) {
    printf("GVM initialize error: invalid format\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  removelist(msgl);
  free(tmp_data);

} /* end initialize */


/****************************************************************************
remove_view: Deallocates the space used by the groupview manager.

****************************************************************************/
void remove_view(view)
link   *view;

{
  if (view->next)
    remove_view(view->next);
  free(view->data);
  free(view);

} /* end remove_view */
```

```
/***************************************************************************
save_grp_view: creates/overwrites the group view file in /tmp directory
          filename is passed as cmd line arg on invocation of gvm
***************************************************************************/
void save_grp_view(grp_name, head_ptr, nmbr, size)
char   *grp_name;
link   *head_ptr;
int     nmbr, size;

{
  char    temp[MAXNUM + 1];
  char    *header, *msg;
  int     msglen;
  FILE  *fd;
  char    *chmod;

  header = CALLOC((2*MAXNUM + 2 + 1), char);
  msg = CALLOC((2*MAXNUM + 2 + size*(MAXLMTSIZE + 1) + 1), char);

  /* create the msg header */
  sprintf(temp,"%i", nmbr);
  strcpy(header,temp);
  strcat(header,"\n");
  sprintf(temp,"%i", size);
  strcat(header,temp);

  /* create the message */
  msg = list2str(head_ptr, header, "\n", "\n");
  msglen = strlen(msg);

  /* open the file */
  fd = fopen(grp_name,"w");

  if (fd == NULL) {
    printf("GVM save_grp_view error: file open\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( ( fprintf(fd,"%s", msg) ) != msglen) {
    printf("GVM save_grp_view error: printf\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* close the file */
  if ( fclose(fd) == EOF) {
    printf("GVM save_grp_view error: file c'ose\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* change the access codes */
  chmod = CALLOC(5 + 2 + 3 + 3 + strlen(grp_name), char);
  strcpy(chmod,"chmod -f 666 ");
  strcat(chmod, grp_name);

  system(chmod);

  /* de-allocate all of the temp storage */
  free(chmod);
  free(msg);
  free(header);

} /* end save_grp_view */
```

```
/* ************************************************************************
 * STATUS TABLE MANAGER (STM)
 *
 * Version: 06 APR 1993
 * Author: David Pezdirtz
 *
 * DESCRIPTION:
 * Waits for a connection and then rc [illegible] message.  Depending on the
 * message type it executes one particu[illegible] action.  Will return a message
 * with size = 0 if empty list.
 *
 * USAGE: stm soc
 *
 ************************************************************************/

/************************************************************************
  Declarations
 ************************************************************************/

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

#define ERROR  0                    /* possible update actions */
#define ADD     1
#define DEL     2
#define UPD     3

typedef char *buffer;

void       create_msg();
void       process_update();
void       process_request();
void       initialize();
void       add();
void       remove_table();
```

```
/************************************************************************
Main :

*************************************************************************/
main(argc,argv)
int          argc;
char         *argv[];
{
  int                         soc,newsoc,clen,msglen;
  int                         action;
  struct sockaddr_un          caller_addr;
  char                        *buf;
  link                        *head = NULL;
  link                        *msgl1;
  int                         group_size = 0;

  if ( argc == 2 ) {
    soc = atoi(argv[1]); }
  else {
    printf("STM usage error: stm soc\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  while( TRUE ){
    clen = sizeof(caller_addr);

    /* Accept connection request from client */
    clen = sizeof(caller_addr);

    if ( (newsoc = accept(soc,(struct sockaddr*) &caller_addr, &clen)) < 0) {
      printf("STM: accept error\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    /* Read message */
    if ( (msglen = readmsg(newsoc, &buf, "#")) < 0 ) {
      printf("STM: read error\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }
    buf[msglen] = NULL;

    /* Determines which message was received */
    action = in_msg_type(buf);

    /* set up the top level list */
    msgl1 = str2list(buf,"\n#");

    switch (action) {
      case STATREQST:
        process_request(newsoc, head, group_size);
        break;


      case UPDSTATUS:
        process_update(&head, msgl1, &group_size);
         break;

      case INITTABLE:
        initialize(&head, msgl1, &group_size);
         break;

      default:
        printf("STM error: invalid msg type\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1);

    } /* end switch */

    /* Dispose of msg list */
    removelist(msgl1);
    free(buf);

    close(newsoc);

  } /* end while */

} /* end main */
```

```
/******************************************************************************
process_update: process the update group view message

usage: process_update(link *head, char *msgl1)
       head - ptr to the qroup view linked list
******************************************************************************/
void process_update(head, msgl1, table_size)
link       **head;
link       *msgl1;
int        *table_size;
{
  int      action, found = FALSE;
  char     *target, *element, *new_status, *current_tgt, *temp_data;
  link     *msgl2, *ptr, *msgl3, *last;

  if ( (getfromlist(msgl1, &element, 2)) == 0 ) {
    printf("STM process_update error: getfromlist(1)a\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  target = CALLOC(strlen(element), char);
  strcpy(target,element);
  msgl2 = str2list(target," #");

  if ( (getfromlist(msgl2, &target, 1)) == 0 ) {
    printf("STM process_update error: getfromlist(1)b\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if ( (getfromlist(msgl2, &new_status, 2)) == 0 ) {
    printf("STM process_update error: getfromlist(2)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* check for delete status */
  if (strcmp(new_status,"delstatus") == 0)
    action = DEL;
  else
    action = UPD;
```

```
/* search for the proper target */
ptr = *head;
last = NULL;

while ((ptr != NULL) && (found == FALSE)) {
  temp_data = CALLOC(MAXLMTSIZE + 9 + 1, char);
  strcpy(temp_data, ptr->data);

  msgl3 = str2list(temp_data," ");

  if ( (getfromlist(msgl3, &current_tgt, 1)) == 0 ) {
    printf("STM process_update error: getfromlist(2)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  if( strcmp(current_tgt, target) == 0 )
    found = TRUE;
  else {
    last = ptr;
    ptr  = ptr->next; }

  removelist(msgl3);
  free(temp_data);

} /* end while ptr & found */

if (found == FALSE) {

  if (action != DEL)
    action = ADD;
  else
    action = ERROR;

} /* end if found */
```

```
switch (action) {
  case ADD:
    add(head,element);
    *table_size = ++*table_size;
    break;

  case DEL:
    /* check for delete of 1st element */
    if (last == NULL)
      *head = ptr->next;                    /* if so, reset the head ptr */
    else
      last->next = ptr->next;/* otherwise remove node from list */

    /* free up the storage used */
    free(ptr->data);
    free(ptr);

    *table_size = --*table_size;
    if (*table_size < 1) {
      *head = NULL; }
    break;

  case UPD:
    strcpy(ptr->data, element);
    break;

  case ERROR:
    printf("STM error: attempt to delete non-existant element\n");

  default:
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$.$$$$$$$$$$$$$\n");
    exit(-1);

} /* end switch */

removelist(msgl2);
free(target);

} /* end process_update */
```

```
/****************************************************************************
process_request : assemble the list into a string and write the message to
            the socket

  output msg format:

  "statustbl"|\n | table size | = |element|sp|status| = |element| ... | # |

****************************************************************************/
void process_request(soc, table, size)
int       soc;
link      *table;
int       size;

{
  char   temp[MAXNUM + 1];
  char   *header, *msg;
  int    msglen;

  /* reserve memory for |header| */
  header = CALLOC((9 + MAXNUM + 2 + 1), char);

  /* reserve memory for |msg| */
  msg = CALLOC((9 + MAXNUM + 2 + size*(MAXLMTSIZE + 9 + 1) + 1), char);

  /* create the msg header */
  strcpy(header,"statustbl");
  strcat(header,"\n");

  sprintf(temp,"%i", size);
  strcat(header, temp);

  /* create the message */
  msg = list2str(table, header, "=", "=");

  strcat(msg,"#");

  msglen = strlen(msg);
```

```
  if ( (writemsg(soc, msg, msglen)) != msglen) {
    printf("STM process_request error: writemsg");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  free(msg);
  free(header);

} /* end process_request */

/****************************************************************************
add : add a node to the end of the current group view.

****************************************************************************/
void add(head, element)
link    **head;
char    *element;
{
  link    *tmp, *ptr;

  tmp = CALLOC(1, link);
  tmp->data = CALLOC(strlen(element) + 1, char);
  strcpy(tmp->data, element);
  tmp->next = NULL;

  if (*head == NULL)
/* check for empty list */
    *head = tmp;
  else {                /* parse to end of list */
    ptr = *head;

    while ( ptr->next != NULL) {
      ptr = ptr->next; }

    ptr->next = tmp;

  } /* end else head */

} /* end add */
```

```
/*****************************************************************************
initialize: initialize the group view.  Delete and removes existing list.

*****************************************************************************/
void initialize(head, list, size)
link **head, *list;
int *size;
{
  link     *msgl;
  int       next_element;
  char     *data, *tdata;

  if (*head != NULL) {
    remove_table(*head);
    *head = NULL; }

  if ( (getfromlist(list, &data, 2)) == 0) {
    printf("STM initialize error: invalid format a\n");
    printf("\07$$$$$$$$$$$$$$`$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  tdata = CALLOC(strlen(data) + 1, char);
  strcpy(tdata, data);

  msgl = str2list(data,"=#");

  if ( (getfromlist(msgl, &data, 1)) == 0) {
    printf("STM initialize error: invalid format b\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  *size = atoi(data);

  next_element = 2;

  while ( (getfromlist(msgl, &data, next_element)) > 0) {
    add(head,data);
    next_element++;
  } /* end while */
```

```
  if ((next_element - 2) < *size) {
    printf("STM initialize error: invalid format\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  removelist(msgl);
  free(tdata);

} /* end


/*****************************************************************************
remove_table: Deallocates the space used by the groupview manager.

*****************************************************************************/
void remove_table(table)
link    *table;

{
  if (table->next)
    remove_table(table->next);

  free(table->data);
  free(table);

} /* end remove_table */
```

```
/***************************************************************************
 * TOKEN POOL MANAGER (TPM)
 *
 * Version: 06 APR 1993
 * Author: David Pezdirtz
 *
 * DESCRIPTION:
 *    Waits for a connection and then reads a message.  Depending on the
 * message type it executes one particular action.
 *
 * USAGE: tpm soc
 *
 ***************************************************************************/


/***************************************************************************
  Declarations
 ***************************************************************************/

#include "gmp.h"
#include "gmputil.c"

typedef char *buffer;

void      process_request();
void      initialize();
void   add();
void      del();
void      remove_table();
char      *strip();

link      *head = NULL, *tail = NULL;
```

```
/***************************************************************************
Main :

 ***************************************************************************/
main(argc,argv)
int         argc;
char        *argv[];
{
  int                     soc,newsoc,clen,msglen;
  int                     action;
  struct sockaddr_un      caller_addr;
  char                    *buf, *data;
  link                    *msgl1;
  int                     pool_size = 0;

  link *ptr;

  if ( argc == 2 ) {
    soc = atoi(argv[1]); }
  else {
    printf("TPM usage error: tpm soc\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  while( TRUE ) {

    clen = sizeof(caller_addr);

    /* Accept connection request from client */
    clen = sizeof(caller_addr);

    if ( (newsoc = accept(soc, (struct sockaddr*) &caller_addr, &clen)) < 0) {
      printf("TPM unix: accept error\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }
```

```
/* Read message */
if ( (msglen = readmsg(newsoc, &buf, "#")) < 0 ) {
  printf("TPM PORT unix: read error\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

buf[msglen] = NULL;

/* Determines which message was received */
action = in_msg_type(buf);

/* set up the top level list */
msgl1 = str2list(buf,"\n#");

switch (action) {
  case TOKPREQST :
    process_request(newsoc, pool_size);
    break;

  case TOKENTOKN:
    if ( (getfromlist(msgl1, &data, 2)) == 0) {
      printf("TPM initialize error: invalid format (a)\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    add(data);
    pool_size++;
    break;

  case DELTTOKEN:
    if ( (getfromlist(msgl1, &data, 2)) == 0) {
      printf("TPM initialize error: invalid format (b)\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    del(data);
    pool_size--;
    break;
```

```
  case INITTPOOL:
    initialize(msgl1, &pool_size);
    break;

  default:
    printf("TPM error: invalid msg type\n");
    printf("TPM error: message recv'd |%s\n",buf);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1);

  } /* end switch */

  /* Dispose of msg list */
  removelist(msgl1);
  free(buf);

  close(newsoc);

 } /* end while */

} /* end main */
```

```
/***************************************************************************
process_request : assemble the list into a string and write the message to the
            socket

  output msg format:

  "tokenpool"\n|size| = |token type| sp |subj| sp |origin| sp | = | ... | # |

***************************************************************************/
void process_request(soc, size)
int         soc;
int         size;
{
  char    temp[MAXNUM + 1];
  char    *header, *msg;
  int     msglen;

  /* reserve memory for |header| */
  header = CALLOC((9 + MAXNUM + 2 + 1), char);

  /* reserve memory for |msg| */
  msg = CALLOC((9 + MAXNUM + 2 + size*(MAXLMTSIZE + 9 + 1) + 1), char);

  /* create the msg header */
  strcpy(header,"tokenpool");
  strcat(header,"\n");
  sprintf(temp,"%i", size);
  strcat(header, temp);

  /* create the message */
  msg = list2str(head, header, "=", "=");
  strcat(msg,"#");

  msglen = strlen(msg);

  if ( (writemsg(soc, msg, msglen)) != msglen) {
    printf("TPM process_request error: writemsg");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
  free(msg);
  free(header);

} /* end process_request */


/***************************************************************************
add : add a node to the end of the current table view.

***************************************************************************/
void add(element)
char    *element;

{
  link    *tmp, *ptr;

  tmp = CALLOC(1, link);

  tmp->data = CALLOC(strlen(element) + 1, char);
  strcpy(tmp->data, element);
  tmp->next = NULL;

  if (head == NULL) {          /* check for empty list */
    head = tmp;
    tail = tmp; }
  else {
    tail->next = tmp;
    tail = tmp; }

} /* end add */
```

```
/**************************************************************************
del : delete a node from the current token pool.  Searches the token pool for
    the token that matches the data element passed.  Ignores the originator
    field for comparison purposes.

**************************************************************************/
void del(element)
char *element;

{
  link   *ptr, *last = NULL;
  int    found = FALSE;
  char   *data, *cur_data;

  ptr = head;

  /* strip off the originator */
  data = strip(element);

  /* search for the proper status */
  while ( (found == FALSE) && (ptr != NULL)) {

    /* strip the originator field from the stored tokens */
    cur_data = strip(ptr->data);

    /* compare the desired data to the current node's data */
    if (strcmp(data, cur_data) == 0)
      found = TRUE;

    else {
      last = ptr;
      ptr  = ptr->next; }

    free(cur_data);

  } /* end while found & ptr */

  free(data);

/* ERROR: requested delete not in list */
if (found == FALSE) {
   printf("TPM delete token: token |%s| not in list\n",element);
   printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
   exit(-1); }

/* check for delete of 1st element */
if (last == NULL)
  head = ptr->next;                     /* if so, reset the head ptr */
else
  last->next = ptr->next;               /* otherwise remove node from list */

/* check for delete of last element */
if (ptr->next == NULL)
  tail = last;

/* free up the storage used */
free(ptr->data);
free(ptr);

} /* end del */
```

```
/***************************************************************************
initialize: initialize the table view.  Delete and removes existing list.

***************************************************************************/
void initialize(list, size)
link  *list;
int   *size;

{
  link     *msgl;
  int       next_element;
  char     *element, *tdata;

  /* remove old list (if any) */
  if (head != NULL) {
    remove_table(head);
    head = NULL; }

  /* get the token pool */
  if ( (getfromlist(list, &element, 2)) == 0) {
    printf("TPM :   tialize error: invalid format (a)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* copy the token pool */
  tdata = CALLOC(strlen(element) + 1, char);
  strcpy(tdata, element);

  /* convert the token pool to a list of pool size & tokens */
  msgl = str2list(tdata,"=#");

  /* get the token pool size */
  if ( (getfromlist(msgl, &element, 1)) == 0) {
    printf("TPM initialize error: invalid format (b)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  *size = atoi(element);

  /* set the offset into the token pool list */
  next_element = 2;

  /* add all tokens in list to local database */
  while ( (getfromlist(msgl, &element, next_element)) > 0) {
    add(element);
    next_element++;
  }  /* end while */

  /* check for errors in processing */
  if (next_element < *size + 2) {
    printf("TPM initialize error: invalid format (c)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  removelist(msgl);
  free(tdata);

}  /* end initialize */



/***************************************************************************
remove_table: Deallocates the space used by the tableview manager.

***************************************************************************/
void remove_table(table)
link *table;

{
  if (table->next)
    remove_table(table->next);

  free(table->data);
  free(table);

}  /* end remove_table */
```

```
/*****************************************************************************
strip: removes the originator from a token.
     returns a ptr to the 'stripped' character string

          typical call:
                    char *string;
                      ...
                    string = strip(token);
                      ...
                    free(string);

*****************************************************************************/
char *strip(data)
char *data;

{
  char *tmpdata, *answer, *tdata;
  link *tlist;

  tmpdata = CALLOC(strlen(data), char);
  strcpy(tmpdata, data);

  answer = CALLOC(strlen(data), char);

  tlist = str2list(tmpdata," #");

  if ( (getfromlist(tlist, &tdata, 1)) == 0) {
    printf("TPM delete error: invalid format (a)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  strcpy(answer,tdata);
  strcat(answer," ");

  if ( (getfromlist(tlist, &tdata, 2)) == 0) {
    printf("TPM delete error: invalid format (b)\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
```

```
  strcat(answer, tdata);

  /* clean up temp storage */
  removelist(tlist);
  free(tmpdata);

  return(answer);

} /* end of strip */
```

DATA CRUNCHING PROGRAM

```
/*************************************************************************
 * Cruncher.c
 *
 * Description: gather all data generated by simpleapp, condense and dump
 *          the output. Requires a group to grow to max size and then shrink back to
 *          a single member.  All failing members MUST be the acwnbr of the host.
 *           i.e. the last member of the group.
 *
 * Usage: crunch <max group size> <file listing members> [<output file>]
 *        NOTE: if <output file> is omitted, the output is sent to stdout
 *
 * Written by: david pezdirtz
 *
 * last revision:  26 Nov 1993
 *
 *************************************************************************/

#include <stdio.h>
#include <sys/filio.h>
#include <sys/time.h>
#include <time.h>

#define CALLOC(n,type) (type *)calloc((unsigned) n,sizeof(type))

main(argc,argv)

int       argc;
char      *argv[];

{
  char  *mbr_file, *output, *cmd, host[10], type[6], type2[5], temp[3],
        filename[10], subj[30], origin[30], grp_view[10];
  FILE  *mbr_fd, *out_fd, *test_fd;
  int      ax_mbr, i, index, skip, total, et, sum;
  float  avg;
  long   sec, usec;

  cmd = CALLOC(128, char);
```

```
switch (argc){
  case 3:
    max_mbr = atoi(argv[1]);
    mbr_file = argv[2];
    output = NULL;
    break;

  case 4:
    max_mbr = atoi(argv[1]);
    mbr_file = argv[2];
    output = argv[3];
    break;

  default:
    printf("CRUNCH usage error: crunch <max # of mbrs> mbrfile [outfile]\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1);
}

/* open the member file */
mbr_fd = fopen(mbr_file,"r");

if (mbr_fd == NULL) {
  printf("CRUNCH: attempt to open mbr file\n");
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

/* open the output file (if necessary) */
if (output != NULL){
  out_fd = fopen(output, "a");
  if (out_fd == NULL){
    printf("CRUNCH: attempt to open output file\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
}
else {
  out_fd = stdout;
}
```

```
fseek(mbr_fd, 0, 0);

,* copy the files to the local directory */
for (index = 1; index <= max_mbr - 1; index ++){

  /* change to the mbr(index) system */
  if (fscanf(mbr_fd, "%s", host) == EOF){
    printf("CRUNCH: reading member file\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  strcpy(cmd, "rcp ");
  strcat(cmd, host);
  strcat(cmd, ":/tmp/timestamp ts");
  sprintf(temp, "%d", index);
  strcat(cmd, temp);

  system(cmd);
}

/* print header to output file */
fprintf(out_fd, "%s %d %s\n\n", output, max_mbr, mbr_file);

/* total # of increasing changes */
for (i = 2; i <= max_mbr; i++){

  sum = 0;

  for (index = 1; index <= i-1; index ++){
    /* open the mbr(index) data file */
    strcpy(filename, "ts");
    sprintf(temp, "%d", index);
    strcat(filename, temp);

    test_fd = fopen(filename, "r");
    if (test_fd == NULL){
      printf("CRUNCH: attempt to open !%s\n", filename);
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }
```

```
/* skip the appropriate # of lines */
for (skip = 1; skip <= (i - index - 1); skip++){

  /* read a line */
  if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
    printf("CRUNCH: reading test file at %s\n", host);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* if agree init --> skip a line */
  if (strcmp(type2, "init") == 0){
    if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
      printf("CRUNCH: reading test file at %s\n", host);
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }
  }

  /* parse the extra data for the agree recv */
  while (fgetc(test_fd) != 10){
   }
  /* skip the comit send */
  if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
    printf("CRUNCH: reading test file at %s\n", host);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

}

/* read the data */
if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
  printf("CRUNCH: reading test file at %s\n", host);
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

et = sec * 1000000 + usec;
```

```
    /* if agree init --> skip a line */
    if (strcmp(type2, "init") == 0){
      if (fscanf(test_fd, "%s%i%i%s%s%s", type, &sec, &usec, type2, subj, origin) == EOF){
        printf("CRUNCH: reading test file at %s\n", host);
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }
    }

    /* parse the extra data for the agree recv */
    while (fgetc(test_fd) != 10){

    }

    /* read the comit send */
    if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
      printf("CRUNCH: reading test file at %s\n", host);
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }

    et = sec * 1000000 + usec - et;

    /* compile the data */
    sum = sum + et;

    /* close the test file */
    if ( fclose(test_fd) == EOF) {
      printf("CRUNCH: attempt to close test file\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1); }
  }
  /* average the data */
  avg = (float) sum / (float) (i - 1) / 1000000.0;

  /* output the data */
  fprintf(out_fd, "i=%d\tavg = %f\n", i, avg);

}
```

```
/* total # of decreasing changes */
  for (i = 1; i <= max_mbr - 1; i++){

    sum = 0;

    for (index = 1; index <= max_mbr - i; index ++){

      /* open the mbr(index) data file */
      strcpy(filename, "ts");
      sprintf(temp, "%d", index);
      strcat(filename, temp);

      test_fd = fopen(filename, "r");
      if (test_fd == NULL){
        printf("CRUNCH: attempt to open !%s\n", filename);
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }

      /* skip the appropriate # of lines */
      for (skip = 1; skip <= (max_mbr + i - index - 1); skip++){

        /* read a line */
        if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
          printf("CRUNCH: reading test file at %s\n", host);
          printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          exit(-1); }

        /* if agree init --> skip a line */
        if (strcmp(type2, "init") == 0){
          if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
            printf("CRUNCH: reading test file at %s\n", host);
            printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
            exit(-1); }
        }

        /* parse the extra data for the agree recv */
        while (fgetc(test_fd) != 10){
        }
  /* skip the comit send */
  if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
    printf("CRUNCH: reading test file at %s\n", host);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

}

/* read the data */
if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
  printf("CRUNCH: reading test file at %s\n", host);
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

et = sec * 1000000 + usec;

/* if agree init --> skip a line */
if (strcmp(type2, "init") == 0){
  if (fscanf(test_fd, "%s%i%i%s%s%s", type, &sec, &usec, type2, subj, origin) == EOF){
    printf("CRUNCH: reading test file at %s\n", host);
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }
}

/* parse the extra data for the agree recv */
while (fgetc(test_fd) != 10){
}

/* read the comit send */
if (fscanf(test_fd, "%s%i%i%s", type, &sec, &usec, type2) == EOF){
  printf("CRUNCH: reading test file at %s\n", host);
  printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
  exit(-1); }

et = sec * 1000000 + usec - et;

/* compile the data */
sum = sum + et;
```

```
      /* close the test file */
      if ( fclose(test_fd) == EOF) {
        printf("CRUNCH: attempt to close test file\n");
        printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
        exit(-1); }
    }
    /* average the data */
    avg = (float) sum / (float) (max_mbr - i) / 1000000.0;

    /* output the data */
    fprintf(out_fd, "i=%d\tavg = %f\n", (max_mbr - i), avg);
  }

  /* close the member file */
  if ( fclose(mbr_fd) == EOF) {
    printf("CRUNCH: attempt to close mbr file\n");
    printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    exit(-1); }

  /* close the output file (if necessary) */
  if ( out_fd == stdout){
    if ( fclose(out_fd) == EOF){
      printf("CRUNCH: attempt to close output file\n");
      printf("\07$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
      exit(-1);
    }
  }
}
```

INITIAL DISTRIBUTION LIST

	Number of Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3. Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4. Professor Shridhar B. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
5. Professor Randy L. Borchardt, Code EC/Bt Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
6. LT David J. Pezdirtz, Jr. 250 Fellows Ave. West Jefferson, Ohio 43162	2