RELIABLE GROUP COMMUNICATION IN DISTRIBUTED SYSTEMS

By

SRIVALLIPURANANDAN NAVARATNAM

B.Eng.(Hons.), The University of Madras, 1983

M.A.Sc., The University of British Columbia, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1987

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 14 - 10 - 87

Abstract

This work describes the design and implementation details of a reliable group communication mechanism. The mechanism guarantees that messages will be received by all the operational members of the group or by none of them (atomicity). In addition, the sequence of messages will be the same at each of the recipients (order). The message ordering property can be used to simplify distributed database systems and distributed processing algorithms. The proposed mechanism continues to operate despite process, host and communication link failures (survivability). Survivability is essential in fault-tolerant applications.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge my appreciation to both my supervisors Dr. Samuel Chanson and Dr. Gerald Neufeld, who have given valuable advice and guidance during the course of this research.

I would also like to thank Ravi who aided with ideas and critisism.

Encouragement from Mehrnaz and Cindy is gratefully acknowledged.

Chapter One
Introduction

1.1 Goal of the Thesis

This thesis is concerned with the design and implementation of reliable one-to-many inter process communication (IPC) mechanism for supporting distributed computations in an environment where certain types of failure could occur. One-to-many IPC (also known as multicast or group communication) refers to an activity by which a single message may be transferred from one process to many other processes which may be in the same or different hosts in the distributed system. The mechanism guarantees that the message will be received by all the operational receivers or by none of them. It also ensures that the messages sent from the senders will be delivered in the same order to all the receivers. The following section describes the motivation behind this work by bringing out examples where a reliable group communication mechanism such as the one proposed is necessary. Section 1.3 briefly describes the underlying system model and the assumptions made in the design of the group communication mechanism. The design philosophy of the proposed group communication mechanism and a general description of the scheme is given in Section 1.4. Section 1.5 reviews previous work and highlights their differences from our proposed mechanism. Section 1.6 concludes this chapter by giving an outline of the thesis.

1.2 Motivation

One of the promises of distributed computing is a more available computing system. To achieve this goal it is necessary to replicate computations

and databases at different hosts which allows a computation to continue to run despite the failures of some of the hosts. In this environment a set of distributed cooperating processes, possibly residing on different hosts, can be viewed as a single logical entity called a process group. The individual processes of a group are sometimes called **members** of the group. Such an architecture allows certain critical resources to be maintained on more than one host and be conveniently shared by client processes with enhanced modularity, performance and reliability [4]. The clients access the members of the group as a single logical entity using the group's logical name. Hence there is a need to communicate the same information to all members of a group. Thus, many applications can benefit from a multiple-destination message transport mechanism such as broadcast and multicast. Broadcast is the delivery of a message to all the destination addresses. Multicast is the delivery of a message to some specified subset of the possible destinations.

Requirement for reliable group communication mechanism arises in applications that are distributed to achieve parallel processing, resource sharing, data availability and reliability. For example, consider an application that updates replicated copies of a distributed database maintained on different hosts as illustrated in Figure 1.1. In order to perform an update to the database, a process first requests the database managers (DBMs) on each host to obtain a lock on the item to be updated. Each DBM will reply with an indication whether or not the lock is available. Here the set of DBMs can be viewed as a process group and the request message can be sent to the DBM group. Clearly this request must be performed reliably in order to assure that all the DBMs will receive the request message. Once it is confirmed that all the locks are acquired, a notification containing the update can then be sent to the DBM group. This notification must also be reliably delivered to each DBM.

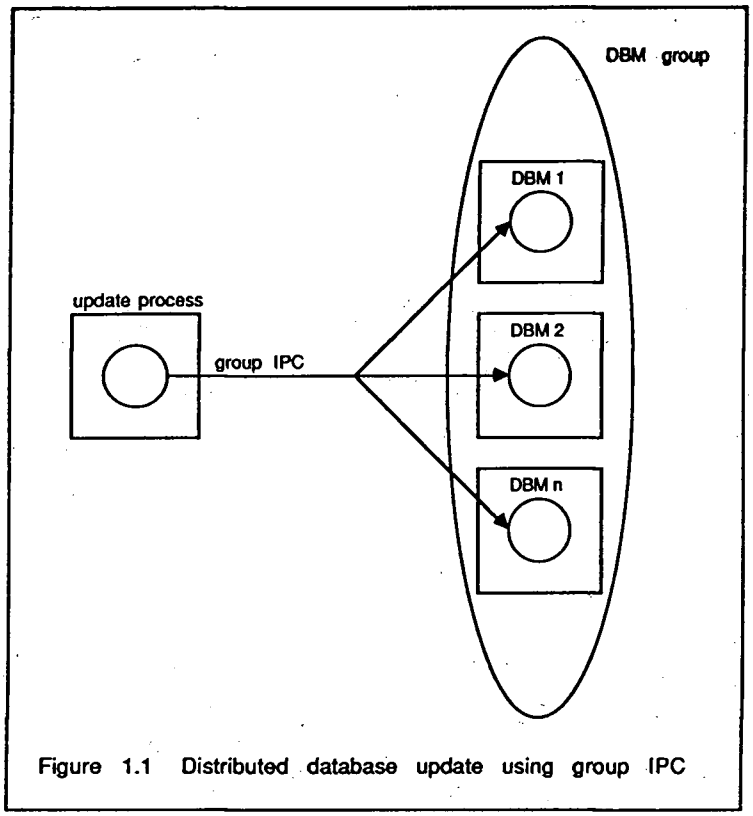


Figure 1.1 Distributed database update using group IPC

In some applications where several processes are interacting with the same group, it is required that the messages sent to the group must not only be delivered to all the members but must also be delivered in the same order. Requiring all the members of a group to receive the messages in the same sequence is stricter (and thus includes higher overhead) than just requiring them to obtain all the messages. However this property is useful in distributed systems. If the members of a group residing at different hosts receive messages in different order, they may not arrive at the same state at all, or may require additional communications to synchronize their states. Furthermore, message sequencing can be used to simplify the design of concurrency control and crash recovery procedures in a distributed database system [3].

1.3 Underlying System Model and Assumptions

The prototype model of the proposed group commnunication mechanism is built on top of the **V Kernel**†, a distributed operating system running on a number of SUN workstations in our Distributed Systems Research Laboratory. These workstations are diskless and connected to a 10 Mbps Ethernet which is a broadcast network. However, the principles of the proposed mechanism is not dependent on the underlying kernel or the network.

In the context of our work two types of processes run in each host; processes responsible for implementing the group communication mechanism and application processes which make use of the group communication mechanism. We assume that the application processes may fail but the processes responsible for implementing the group communication mechanism never fail unless the host machine itself fails. We also assume that when processes or hosts fail they simply cease execution without making any malicious action (i.e., fail stop) [16]. If the host at which a failed process was executing remains operational, we assume that this failure is detected by the underlying operating system and that all the interested parties are notified [14]. On the other hand if the host itself fails, all the processes executing in it fail and processes at other hosts can detect this only by timeouts. Furthermore, we assume that the underlying system provides a reliable one-to-one message transport protocol. In other words, error detection and correction mechanisms (such as checksum, timeout and retransmission) exist which guarantee a unicast message to be delivered to it's destination free of errors.

In the environment where our proposed group communication mechanism is built, no information survives host failures. Since hosts are diskless there is no possible recovery from stable storage. Therefore the case of a process in a host

---

† The semantics of IPC facilities provided by the V Kernel is described in Appendix A.

receiving a message before host failure and one where the host fails before the message is delivered to it are indistinguishable. Thus, our group communication mechanism can only guarantee that all operational members of a group will receive all the messages in the same order.

1.4 General Design Philosophy of the Proposed Group Communication Mechanism

The design of the group communication mechanism should be general and not dependent on specific characteristics of the group or functions available from the underlying network. For example the group may be static or dynamic depending on whether their membership list may change. The underlying hardware may or may not support broadcast and multicast facilities. Consider the case where the underlying hardware supports only a single-destination message transport mechanism (unicast). In this case, delivery of a message to the group can be achieved only by maintaining the list of members in the group, and sending the message to individual members using one-to-one IPCs. However if the underlying hardware supports broadcast and multicast then the members of a group can subscribe to a particular multicast address. A message intended for the group can be sent to this address and only those hosts where one or more members of this group reside will read the message.

Broadcast networks such as Ethernet gives the impression that they provide reliable delivery in the hardware; but in reality they do not. Messages transmitted in these networks are available to all the receivers, but some or all of the receivers may lose messages. Some examples [15] of how this may happen are given below:

1. The buffer memory might be full when a message arrives at the interface unit.

2. The interface unit might not be monitoring the network at the time the message is delivered.
3. In a contention network, an undetected collision that affects only certain network interface units could cause them to miss a message.

Unlike the reliable transport of the unicast packet where the sender can retransmit the message packet until the receiver acknowledges, it is hard to support reliable transport of multicast packets unless the number and identity of the group members are known. If the membership list is maintained, then the message can be multicast to the members in a datagram fashion first and members whose acknowledgements are not received within a fixed time interval can be sent the message again on a one-to-one basis.

Thus, for reliable delivery of messages to all members of a group, some coordination mechanisms are needed to maintain the group membership list. For the static group where the group membership never changes, the coordination mechanism can be built into the underlying system. However for the dynamic group where members may join or exit at any time, a **group manager** is necessary to maintain the membership list. However, this scheme will be render ineffective if the host where the group manager is executing fails. One solution to this problem is to replicate the group manager at all member sites and select a new group manager among these replicas in case of failures. Thus, a group will have one primary group manager (or simply primary manager) and zero or more secondary managers. Mechanisms for selecting the primary manager include token scheme [3], succession list [10] and election [12].

In addition to reliable delivery of messages to all members of the group, the group communication mechanism must also ensure that messages are

delivered in the same order to all the members. In a system with a single sender and many receivers, sequencing messages to all of the receivers is trivial. If the sender initiates the next multicast transmission only after confirming that the previous multicast message has been received by all the members, then the messages will be delivered in the same order. This is illustrated in Figure 1.2. On the other hand, in a system with many senders and a single receiver the messages will be delivered to the receiver in the order in which they arrive at the receiver's host. Ordering in this case is a non-problem as illustrated in Figure 1.3.

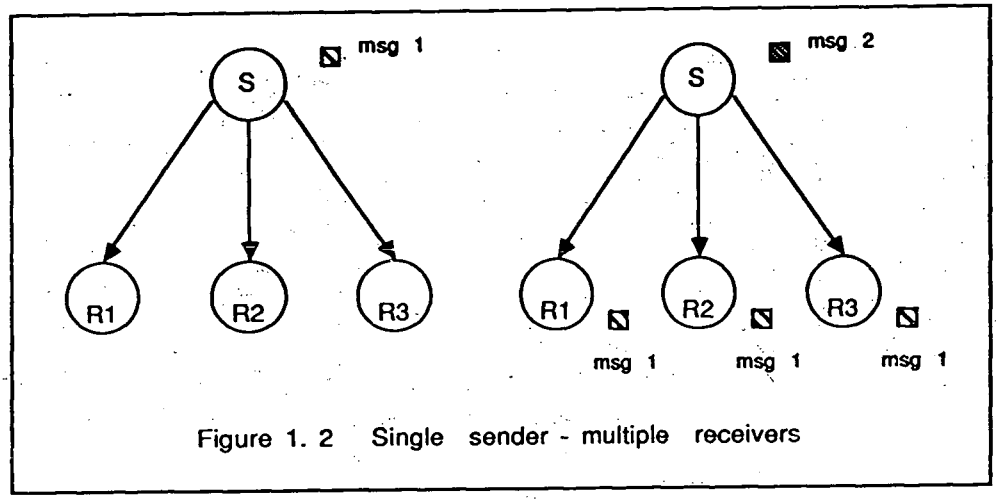


Figure 1. 2 Single sender - multiple receivers

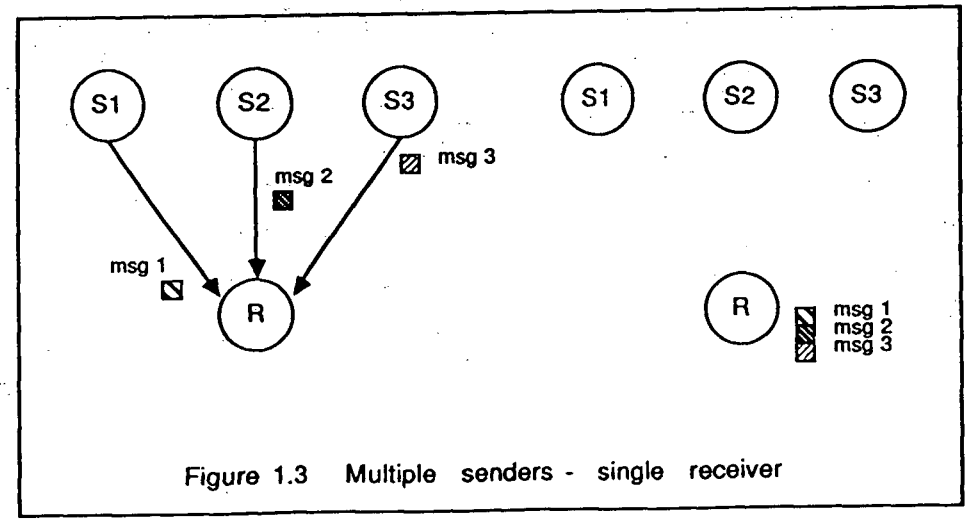


Figure 1.3 Multiple senders - single receiver

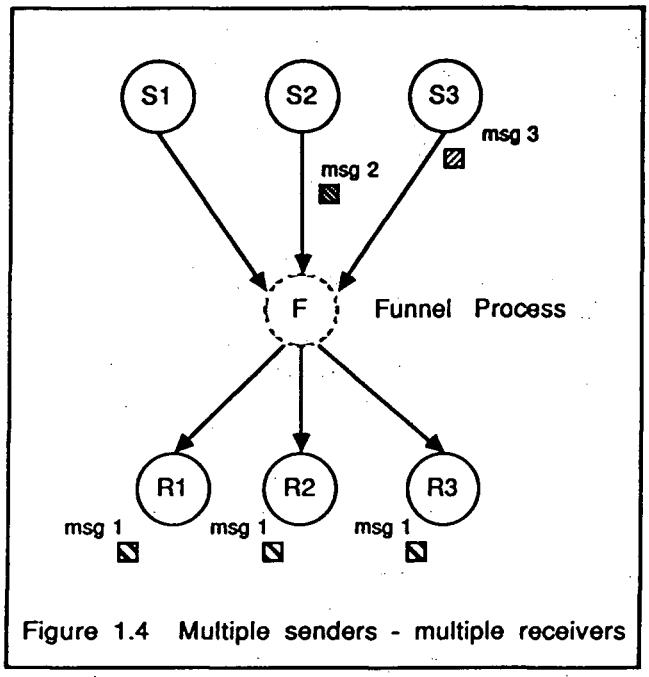


Figure 1.4 Multiple senders - multiple receivers

In general, group communication mechanism must operate between many senders and many receivers. In such a system, a message sent from a sender may arrive at a destination before the arrival of a message from another sender; however this order may be reversed at another destination. A solution [4] to order messages in such a system is to make it appear as a combination of two simple systems, one with many senders and a single receiver, the other with a single sender and many receivers. Therefore the senders will send their messages to a single receiver which then transmits the messages to the rest of the receivers in an orderly fashion. Thus, the single receiver acts as a funnel process as shown in Figure 1.4. This idea can be incoorporated into our design without any additional cost because the group managers which we use to guarantee reliable delivery can be used as the funnel processes as well. Thus in our scheme, the senders will send the messages intended for a particular group to it's primary manager which will then reliably and orderly transmit them to the members of this group.

1.5 Related Work

Although group communication has received considerable attention [1,3,4,7,8,9], only a few distributed systems have actually implemented such facilities. We have chosen to look at four such projects which we consider relevant to our work.

V system [7] defines reliable group communication to mean that at least one member of the group receives the message and replies to it. Each host has information only about local members of the groups. This information includes the identifiers of the local members and their group addresses. So when messages are sent to a group address, hosts where members of this group are executing will receive it and deliver it to the members. The underlying kernel will retransmit the packet until at least one of the members of the group acknowledges the message. Therefore the V Kernel supports a very basic group communication mechanism to transport a message to mutiple processes; additional properties such as reliability and order have to be built on top of it.

Cristian et al. [8] proposed a protocol for the reliable and ordered delivery of a message to all hosts in a distributed system (i.e., broadcast) whereas our focus is on the delivery of a message to a set of processes, several (or all) of which could reside on a single host. Their protocol is based on a simple information diffusion technique. A sender sends a message on all it's (outgoing) links and when a new message is received on some (incoming) links by a host, it forwards that message on all other (outgoing) links. After the reception of the message at a host, it's delivery is delayed for a period of time determined by the intersite message delivery latency. The messages are time stamped to enable order delivery and to detect duplicates. The performance of this protocol is dependent on the accuracy with which the clocks are synchronized and the

operating system's task scheduling mechanism which is responsible for scheduling the relay task which relays an incoming message to the adjacent hosts.

Chang et al. [2] proposed a protocol which, like Cristian's work, is responsible for the delivery of a message to all the hosts in the distributed system. However their philosophy is similar to our's where the messages are funneled through a coordinator called token host. Senders send their messages to the token host which then transmits the message to the rest of the hosts. The protocol places the responsibility on the receiver hosts for reliable delivery. The token host sequences the messages and transmits them to the rest of the hosts in a datagram fashion. If a host misses a sequence number then it sends the token host a negative acknowledgement for the missing message. The token host is rotated among the operational hosts to provide reliability and resiliency.

Birman's ISIS system [1] supports reliable group communication mechanism similar to our's. However, to ensure the order property, the messages are not funneled through a coordinator, instead a two-phase protocol is used. The protocol maintains a set of priority queues for each member, one for each stream of messages, in which it buffers messages before placing them on the delivery queue. When a message is received by a member, it temporarily assigns this message an integer priority value larger than the priority value of any message that was placed in the priority queue corresponding to the message's stream. Each member sends back this priority value to the sender. The sender collects all the replies and computes the maximum value of all the priorities received. It sends this value back to the recipients which assign this priority to the new message and place it on the priority queue. The messages are then transferred from the priority queue to the delivery queue in order of increasing priority. This guarantees order. However, the sender has to reliably communicate with the

members twice before the message is delivered.

All the above works make the same assumptions as outlined in Section 1.3. In addition to these, they also assume that the underlying network never partitions. We do not make such an assumption. In our scheme if the network partitions resulting in subgroups of sites, communication within these subgroups remains possible. When the networks remerge again, the proposed mechanism merges these subgroups to form a single group.

1.6 Outline of the Thesis

The rest of the thesis is organized as follows. Chapter Two examines the properties of reliable group communication mechanism. Issues related to reliability, namely, availability, order, atomicity and survivability as applicable to our group communication mechanism are also discussed. In Chapter Three we describe our group communication mechanism in detail and discuss how the scheme works in the presence of failures. Chapter Four describes the implementation details and the performance of the proposed mechanism. Chapter Five concludes this work.

Chapter Two

Properties of a Reliable Group Communication Mechanism

An important property of group communication mechanisms is reliability. Many researchers use the term *reliability* to mean full delivery of the message, i.e., assuming the sender does not fail in the middle of transmission, messages are delivered to all members of the group [4,7]. However, in this thesis, we consider a group communication mechanism reliable only if it satisfies the two aspects of reliability: full delivery and correctness. Issues related to correctness are order, atomicity and survivability. The order property guarantees that messages sent from all the senders are delivered in the same order to all operational members of the group. Atomicity ensures that every message transmitted by a sender is either delivered to all operational members of the group or to none of them. Survivability is a measure of how well the mechanism is able to tolerate and recover from failures. The following section briefly discusses the concept of full delivery and in Section 2.2, issues related to the correctness of the group communication mechanism in a distributed system running on a cluster of diskless workstations are discussed. Section 2.3 outlines the group send primitives provided by the proposed group communication mechanism which satisfies the above properties. Section 2.4 concludes this chapter.

2.1 Full Delivery

Full delivery ensures that a message sent to a group will be delivered to all operational members provided the sender does not fail in the middle of the transmission. If the underlying network supports broadcast or multicast facilities then one way to implement full delivery is for the sender to broadcast the message to the group first in a datagram fashion and later transmit the

message individually to the members which did not receive the message the first time using one-to-one IPC. However, if the underlying network supports only unicast then the sender may adopt the brute-force method of sending the message to each member individually using one-to-one IPC. Therefore as long as the underlying system supports one-to-one IPC which guarantees reliable delivery of a message to it's destination, the group communication mechanism can ensure the full delivery property.

2.2 Correctness

In addition to full delivery, we also attempt to ensure the *correctness* of the proposed group communication mechanism. In this section we will discuss the issues related to the correctness property.

2.2.1 Order

In a distributed system where processes coordinate their actions by sending messages to one another and do not use a global clock for synchronization, events can only be partially ordered in terms of the **happened before** relation [13]. If we assume that sending or receiving a message is an event in a process then we can define the **happened before** relation denoted by -> as follows.

1. If p and q are events in the same process and if p occurs before q, then p->q.
2. If event p corresponds to sending a message by one process and event b corresponds to receiving the same message by another process, then p->q.
3. If p->q and q->r then p->r. Two distinct events p and q are concurrent if p->q and q->p.

Another way of viewing the definition of **happened before** is to say that p->q means that it is possible for event p to causally affect event q [13]. Consider an event b corresponding to sending a message B to a group G. Let b' be an event of sending a message B' to the same group G. If the two events are initiated by the same process and if b occurs before b' then b->b'. Therefore all the members of the group will receive the messages B and B' in the same order: B first and B' second.

However if the events b and b' are initiated by two different sources S and S' respectively, one cannot causally relate the events b and b' in general unless both sources send their messages to a single receiver R, which then transmits them to the members. Thus using the above example, assume that c is the event corresponding to R transmitting the message B to the members of the group after it has received it from source S. The **happened before** relation denoting this action is given by b->c. Similarly, if c' is the event of R sending the message B' to the group members after R has received it from source S', then b'->c'. Since events c and c' occur in the same process R, and R can only

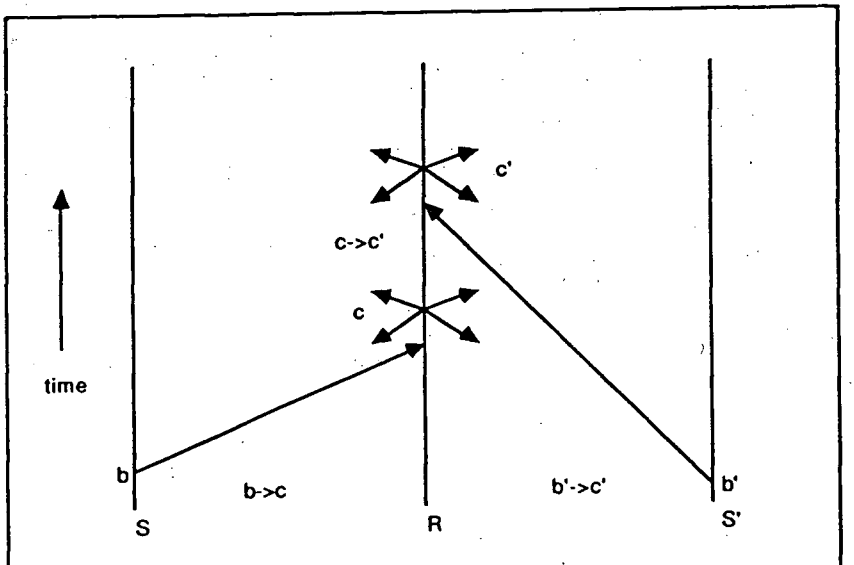


Figure 2.1 Happened before relation for ordered delivery

send a message at a time, events c and c' cannot occur simultaneously. If c->c' then b->b' else if c'->c then b'->b. The **happened before** relation c->c' is illustrated in Figure 2.1. Thus, by using a single receiver to first receive the messages from the sources and then transmitting to the members, one can guarantee that the messages will be delivered to the members in the same order.

For some applications it is not sufficent that messages from different senders are received in the same order but it is also necessary that this order be the same as some predetermined one. Birman [1] gives a following example of such a condition. Consider a process p which instructs a group of devices with the message "place wine bottles under taps" and process q that orders the same group of devices with the message "open taps". Clearly, it is important that the first message be delivered to all members of the group before the second one. One way this can be implemented in a distributed system that does not use a global clock for synchronization is to require the process p to send a message to process q after the wine bottles have indeed been placed under the taps. This message causally relates the group message from p to that from q. Our proposed group communication mechanism does not provide this facility which is left to the application programs.

2.2.2 Atomicity

Atomicity ensures that every message sent to a group is either delivered to all operational members of the group or to none of them. It is important to distinguish the difference between full delivery and atomicity. Full delivery ensures the delivery of messages to all members as long as the sender does not fail during the message transmission. However if the sender fails in the middle of the message transmission, it is possible that some of the members have not

received the message resulting in a partial delivery. Partial delivery is harmful in many applications. Consider an application using the group communication mechanism to implement a replicated file service. Here, all the file servers will belong to a group and updates are sent to this group using the group communication mechanism. If an update is not delivered to any of the file servers, the files at the servers will still be consistent with one another. However, if an update is delivered only to some file servers then some files will be updated while others are not, resulting in inconsistencies. Therefore if a message is delivered to at least one operational member of a group then the group communication mechanism must make sure that this message will be delivered to the rest of the operational members as well. Atomicity property guarantees such an action.

2.2.3 Survivability

Survivability guarantees continuous operation despite failures. In the proposed group communication mechanism, the failure of the primary manager in the middle of a message transmission will result in a partial delivery. In order to survive such failures, the primary manager is replicated in all the member sites. In case the primary manager fails, a new primary manager is selected from among these replicas using some selection mechanism. The new primary manager must finish any incomplete message transmission initiated by the failed primary manager before resuming normal operation. Failures may occur during the selection of new primary manager, and the network may partition. The survivability property must ensure that the group communication mechanism will survive any such failures and still provide order and atomicity to message transmission.

2.3 Outline of the Group Send Primitives

In this section, we first summarize the properties of the proposed group communication mechanism and outline the two primitives provided by the proposed mchanism.

Our reliable group communication mechanism satisfies the following properties.

1. A message sent to a group must be delivered to all operational members of the group or to none of them.
2. If message B is sent to a group before message B' by the same sender, then if B' is received, B is also received.
3. If two messages B and B' are sent by the same sender to the same group, then the messages are received by the members of the group in the same order as they were initiated.
4. If two messages B and B' are sent by two different senders to the same group then the messages are received by all the members of the group in the same order, either B first and B' or B' first and B.

Although the last property is essential in many applications, some applications do not require an order to be enforced between two messages as the outcome of one may not causally affect the other. For example, consider a computation which updates copies of two different variables V1 and V2 maintained by the members of a group. Assume message B broadcast from a source in the computation is responsible for updating V1 and B' from another source in the same computation is responsible for updating V2. In such a scenario it is not necessary that both messages be received by the members of the group in the same order as updating the variable V1 does not have any effect on V2 and vice versa. The only requirement here is that all members must receive the updates or none of them should receive the updates.

Since the overhead in enforcing the *order* property is non-trivial, our group communication mechanism provides two primitives. One guarantees delivery of the messages in the same order to all members of a group and the other guarantees only atomicity, but messages may be delivered in some arbitary order. The former type of message transmission is known as OGSEND (Ordered Group Send). OGSEND messages will be delivered in the same order to all members of a group or to none of them. OGSEND message transmission is initiated by invoking **ogsend**(*msg, gid, msgtype*) where msg is a pointer to the message to be transmitted and gid is the identifier of the group to which the members belong. The second type of transmission UGSEND (Unordered Group Send) does not guarantee ordered delivery but ensures atomicity. UGSEND message transmission is initiated by invoking **ugsend**(*msg, gid, msgtype*). In both the primitives if the IMMEDIATE_REPLY_BIT is not set in msgtype, then the process which invokes these primitives will be unblocked only after the message is delivered to all the operational members of the group. Otherwise, the sending process may be unblocked before the message is indeed delivered to the members of the group as explained in Section 3.2.

2.4 Chapter Summary

Properties of a reliable group communication mechanism includes full delivery and correctness. Full delivery ensures that a message sent to a group will be delivered to all operational members provided the sender does not fail in the middle of the transmission. Issues related to correctness are order, atomicity and survivability. In the proposed mechanism ordering is achieved by funneling the messages through a single process. Atomicity guarantees that if the message is delivered to at least one operational member of a group, then it will be delivered to the rest of the operational members as well. Survivability ensures

continous operation despite host, process and network failures. The proposed mechanism provides two group send primitives **ogsend** and **ugsend** with the above properties.

Chapter Three
Design of the Proposed Group Communication Mechanism

This chapter describes the design philosophy of the proposed group communication mechanism. We have seen in Chapter Two that a reliable group communication mechanism requires some form of coordination to ensure full delivery and the correctness properties. Thus, in the proposed group communication mechanism, each group has a primary manager process which maintains the membership list of the group and also acts as a funnel process for the messages transmitted to the members of the group.

In order to ensure survivability in case of primary manager failure, the primary manager is replicated in all the member sites. We call these replicas **secondary managers**. These secondary managers do not take part in any group management activities which are only carried out by the primary manager. Secondary managers act as backups, so that in case of primary manager failure, one of the secondary managers will take over as the new primary manager. Section 3.1 describes the activities of the primary and the secondary managers as well as the group state information maintained by them. In Section 3.2, the design of the group send primitives outlined in Section 2.3 is discussed. Section 3.3 describes the failure detection and recovery procedures for group members, primary manager and secondary managers. Obviuosly, failure of the primary manager is more serious than the failure of the secondary managers. A new primary manager must be selected from among the secondary managers. There are several schemes proposed in the literature to select a leader in an environment such as ours. Section 3.4 gives an overview of the selection schemes and presents a new scheme based on finite state model used in our proposed group communication mechanism. Section 3.5 deals with a different kind of

failure, i.e., failure due to network partition. Section 3.6 concludes this Chapter.

3.1 Primary and Secondary Group Managers

Each group has a primary manager and zero or more secondary managers. When a new group is created, a primary manager for this group is also created in the same host by the underlying group mechanism. When a member from a different host joins the group, and a secondary manager for this group does not already exist on the joining member's host, a secondary manager for this group will also be created on that host. The primary as well as the secondary managers maintain the process identifiers (pids) of the members of the group local to their respective hosts in the **local group member list.** Also the primary and secondary managers maintain the pids of all the managers for the group in their **manager member lists.** When a new secondary manager is created, the primary manager's manager member list is copied into the new secondary manager's manager member list. The primary manager then updates it's manager member list with the pid of the new secondary manager and informs all the secondary managers of the group to update their lists as well.

When a member joins the group from a host where the primary or a secondary manager for this group already exists, the pid of the new member is simply added to the local group member list. Although group membership information is distributed across all the hosts, the primary and secondary managers maintain information about those members of the group executing locally (i.e., local members). Thus, when a message is sent to a group, the group communication mechanism must make sure that the message is delivered to all the group managers each of which will then deliver the message to it's local members. This requires less space, less network traffic and reduced code complexity compared to the case of replicating the entire membership information

in the primary manager and in all the secondary managers.

3.2 Design of the Group Send Primitives

This section presents the design details of the group send primitives **ogsend** and **ugsend** (outlined in Section 2.4), which make use of the primary manager and the secondary managers to provide the *reliable* properties discussed in Chapter Two.

3.2.1 Ordered Group Send (OGSEND) Primitive

Messages sent to a group by invoking the **ogsend** primitive are delivered in the same order to all members of the group. OGSEND messages to a group are first received by the primary manager for that group, which will then sequence the messages in the order they were received and send them to it's local members and to the secondary managers of the group. When the secondary managers receive the message, they deliver it to their local members. After ensuring that the message is received by all the secondary managers (i.e., after all the secondary managers acknowledge the receipt of the message), the primary manager unblocks the sender which has remained blocked after invoking **ogsend.** A high level description of the message transmission activity of the primary manager is given in Figure 3.1. The primary manager will receive a new message for transmission only after it has completed the delivery of the previous message. Messages arrived at the primary manager's host while it is not ready to receive will be queued first-in-first-out (FIFO) in the primary manager's message queue by the underlying system. When the primary manager is ready to receive a message, the underlying system will deliver the first message in the message queue (if any) to it. Since the message queue is FIFO, messages will be delivered to the primary manager in the order they arrive.

```
Type : group manager
Task : sending a group message

FOREVER DO
    Begin
      Receive (from source)
      Send (to all members)
      Reply (to source)
    End
```

Figure 3.1 Group manager's message transmission

The method of transmitting a message from the primary manager to secondary managers depends on the functionality of the underlying network architecture. If the network only supports unicast facility, then the primary manager can send the message to the individual secondary managers using one-to-one IPC. However, if the network also supports broadcast facility then the message can be first multicast to the group's secondary managers in a datagram fashion. The primary manager then waits for a specific time period for acknowledgements. If acknowledgements are not received from some secondary managers at the expiration of the time interval, the primary manager resends the message to these secondary managers using one-to-one IPC. This allows the primary manager to exploit the positive acknowledgement and retransmission properties of the one-to-one IPC for reliable delivery and to determine the failures described in Section 3.3.

Let's assume that it takes an average of T1 seconds for a message from a primary manager to be delivered to the secondary managers, an average of T2 seconds for this message to be processed by a secondary manager, and an average of T3 seconds for an acknowledgement from a secondary manager to be received and processed by the primary manager. Thus, if the group has **n** secondary managers, then it will take (T1 + T2 + **n**T3) seconds for all the

secondary managers to acknowledge for primary manager's message (assuming no resends).

If T2 includes the time required by a secondary manager to deliver the message to it's local members and to receive their acknowledgements (i.e., one-to-one IPC), then T2 is an application dependent quantity. Thus, if some local members take a long time to process the sender's message, then their secondary manager cannot send an acknowledgement to the primary manager immediately which in turn cannot unblock the sender. However, if the sender wishes that all the members should receive the message before the primary manager unblocks it, then there is no other alternative than to wait for the secondary managers to acknowledge after guaranteeing that the message is received by all of their local members.

On the other hand, it may be acceptable to unblock the sender after the secondary managers have received the message without waiting for acknowledgements from all the group members. In this case, the secondary managers can send acknowledgements to the primary manager without waiting to deliver the message to their local members. The primary manager then unblocks the sender. The secondary managers will queue the messages in the delivery queue for the local members and when the local members are ready to receive, they can obtain the messages from the delivery queue.

The implementation provides flexibility for the applications to specify which scheme they prefer using the IMMEDIATE_REPLY bit of the *msgtype* parameter in **ogsend** and **ugsend** primitives.

The mechanism described so far is inadequate to guard against duplicate messages. For example, acknowledgements for the datagram from some secondary

managers may not be received by the primary manager within the specific time period if the message or the acknowledgement is lost. Thus, when the primary manager resends a message, the secondary managers which did not receive the message the first time around will be receiving the right messasge. However, those secondary managers whose acknowledgements were lost will be receiving a duplicate message. Therefore, the group communication mechanism should incorporate some duplicate detecting scheme. In our proposed mechanism, the primary and secondary managers use **transaction identifiers** to detect and discard duplicate messages. Transaction identifiers are simply integer values. The primary manager maintains a variable called ogsend-send-seq-no **(ossno)** which keeps track of the next OGSEND message's transaction identifier. When a OGSEND message transmission is initiated, the primary manager assigns the **ossno** to the message and transmits it to the secondary managers. The **ossno** is then incremented by one ready to be used with the next OGSEND message. On the receiving end, the secondary managers maintain a variable called ogsend-receive-seq-no **(orsno)** to keep track of the transaction identifier of the next incoming OGSEND message. When an OGSEND message is received by a secondary manager, the **ossno** and **orsno** are compared and depending on their values, the message is either delivered to the local members or discarded.

The OGSEND primitive therefore guarantees the delivery of the messages to all operational members of the group in the same order. Failure detection and recovery procedures of OGSEND message transmission in the event of primary manager failure will be discussed in section 3.3.

3.2.2 Unordered Group Send (UGSEND) Primitive

Although many applications require that messages be delivered to all the members of a group in the same order, some applications do not require such

strict ordering with it's attendant overhead. For these applications, the proposed group communication mechanism provides a primitive called **ugsend.** Messages transmitted by invoking **ugsend** are guaranteed to be delivered to all the members of the group but in some arbitary order.

Unlike ordered delivery, unordered delivery does not require that the messages be funneled through a single receiver. Thus, we have multiple senders and multiple receivers. If each sender maintains a list of all the receivers' pids then every sender can participate in the message transmission activity. Even though the messages from senders can be guaranteed to be delivered to all the members, they may not be delivered in the same order.

In the proposed group communication mechanism, each secondary manager has information about the primary manager as well as all the secondary managers (manager member list). Thus, every secondary manager can initiate a message transmission similar to the primary manager's OGSEND transmission. For example assume that c is the event corresponding to secondary manager C transmitting the message M to the members of the group after it has received it from sender S. Similarly, c' is the event of secondary manager C' transmitting the message M' to the members of the group after C' has received it from sender S'. Since events c and c' occur at different processes, it is possible that both events may occur at the same time. Under such circumstances, message M will be delivered before message M' to some members of the group and in the reverse order to the rest of the members.

Thus, when applications invoke the **ugsend** primitive to send messages to a group, the group communication mechanism first checks to see whether there is a manager for this group available in the sender's host. If there is, the message will first be sent to it which will then transmit it to the rest of the

managers, each of which in turn deliver the message to thier local members. However if a local manager does not exist, then the message is sent to the primary manager for the group which then transmits it to it's local members and to the secondary managers for the group.

Similar to OGSEND transmission, UGSEND transmission needs a mechanism to detect duplicates. In our scheme, when the primary or a secondary manager for a group transmits a UGSEND message, a ugsend-send-seq-no **(ussno)** is assigned to the message. The recipients will have the corresponding ugsend-receive-seq-no **(ursno)**. When a manager for the group receives a UGSEND message, the **ussno** and **ursno** are compared and depending on their values the message will either be delivered to the local members or discarded.

3.3 Failure Detection and Recovery Procedures

To ensure that the group communication mechanism provides reliable service, the survivability property must be guaranteed. Survivability is a measure of how well a system can tolerate and recover from failures. Our discussion in this section will focus on two aspects of failures: process failures and host failures. We have assumed that application processes such as group members may fail, but operating system processes such as primary or secondary managers which are used to implement the group communication mechanism are well debugged and do not fail unless the host machine itself fails. When the host fails all the processes in it fail. Therefore host failures are more serious than process failures. Suppose the host fails while a primary or a secondary manager executing in it is in the middle of a message transmission, it is possible that some of the members will not receive the message resulting in a partial delivery. The following section briefly describes failures of group members. In Section 3.3.2, failure detection and completion of any incomplete UGSEND

message transmission in the event of secondary manager failure will be discussed. The case of primary manager failure is described in Section 3.3.3.

3.3.1 Group Member Failure

Failure of a group member does not affect group communication activities in the other operational group members. Our group communication mechanism guarantees that all the operational group members of the group will receive the messages sent to them. The failure of a member is detected when a primary or a secondary manager tries to deliver a message to the failed member using a one-to-one IPC (refer to Section 3.2.1). On detecting the failure of one of it's local members, the primary or the secondary manager simply removes the failed member's pid from it's local group member list. After the removal, if the local group member list maintained by a secondary manager becomes empty, then this manager ceases execution. On the other hand if the primary manager's list becomes empty and it's manager member list is also empty, then the primary manager ceases execution and the group is considered nonexistent.

3.3.2 Secondary Manager Host Failure

If a secondary manager fails, the primary manager has to detect this and finish any incomplete UGSEND message transmission initiated by the failed secondary manager. To detect the failure of secondary managers, the primary manager has many options. All these schemes exploit the positive acknowledgement property of one-to-one IPC to determine process failure.

In one scheme, the primary manager will detect the secondary manager failure in the next OGSEND or UGSEND message transmission, or a transmission of a group view update such the creation of a new secondary manager for the group. When the primary manager tries to send a message to

the individual secondary managers using one-to-one IPC, if a secondary manager has failed then the underlying system will inform the primary manager that it is trying to send a message to a nonexistent process (see Appendix A).

Another scheme is to create a **vulture** process to look for the failure of a secondary manager. A vulture process is a light weight process created by the primary manager at it's host. Since the proposed mechanism is built on top of the V Kernel, the vulture process takes advantage of the **ReceiveSpecific** IPC primitive provided by the underlying system to detect secondary manager failures. A high level description of the vulture process to detect the failure of secondary manager sm(i) is shown in Figure 3.2. The vulture will be received blocked on secondary manager sm(i) as long as the latter does not send any messages to it. However, if sm(i) fails, the underlying kernel in the vulture's host will unblock the vulture and notify it that it is trying to receive a message from a nonexistent process (see Appendix A). The vulture then informs the primary manager about the failure of secondary manager sm(i). For this scheme to work, the primary manager has to create **n** vultures if there are **n** secondary managers in the group.

Rather than creating a vulture process for each secondary manager, the primary manager may create a single process called a **prober** to probe the liveliness of it's secondary managers. The prober periodically sends probe message

```
Type : vulture
Task : detect failure of primary manager (pm)

Begin
  ReceiveSpecific (from pm)
  Send ("pm failed" to secondary manager)
End
```

Figure 3.2 Vulture process

ARE_YOU_ALIVE to each secondary manager. The probe message uses one-to-one IPC's to which the secondary managers will reply with I_AM_ALIVE messages. If a secondary manager fails then the underlying system will inform the prober that it is trying to send a message to a nonexistent process and the prober will notify the failure to the primary manager. A high level description of the prober process is given in Figure 3.3.

The first scheme may take longer, before the primary manager detects a secondary manager has failed. This is due to the fact that the primary manager depends on it's next message transmission which may not happen for a long time to detect the failure. The second scheme is expensive because each secondary manager needs a separate vulture process. The third option is less expensive than the second scheme since only a single process has to be created on the primary manager's site to detect the failures of all the secondary managers. It is also faster than the first scheme because it does not depend on the next message transmission. Our prototype implementation uses the third scheme.

```
Type : prober
Task : detect failure of secondary managers
FOREVER DO
    Begin
      For i = 1 to n Do
        Send (ARE_YOU_ALIVE to sm(i))
        If reply != I_AM_ALIVE Then
           Send (sm(i) failed to primary manager)
        Sleep (for a specified time period)
    End
n = number of secondary managers
```

Figure 3.3 Prober process

Once the failure of a secondary manager is detected, the primary manager has to delete the failed secondary manager's pid from it's manager member list and inform the rest of the operational secondary managers to do the same in order to maintain a consistent group view. However, before doing this, the primary manager must finish any incomplete UGSEND message transmission initiated by the failed secondary manager. The primary manager requests all the secondary managers to send to it their last UGSEND message received. If the returned messages as well as the last message received by the primary manager have the same transaction identifier value, then the failed secondary manager has either successfully completed it's last UGSEND message transmission activity or no member has received it's last UGSEND message. Either of these outcomes assures *atomicity*. However if there is a discrepancy among the transaction identifier values†, then the primary manager takes the message with the highest transaction identifier and transmits it to the secondary managers. Those secondary managers that have already received the message simply discard the duplicates, but others receive the message and deliver it to their local members. Once this message retransmission activity is completed, the primary manager deletes the failed secondary manager's pid from it's manager member list and informs the rest of the operational secondary managers about the failure.

3.3.3 Primary Manager Host Failure

The primary group manager fails when the host in which it is executing fails. Primary manager failure is more serious than secondary manager failure. If the primary manager fails, OGSEND activities cannot be carried out and new members cannot join the group from a host where a secondary manager for this

---

† The transaction identifier values will differ by at most one.

group does not reside. Also, failure of secondary managers cannot be detected and incomplete message transmission activities initiated by the failed secondary managers cannot be finished. Even though the operational secondary managers may be able to participate in UGSEND message transmission, one cannot guarantee atomic delivery. Thus, a group cannot exist without a primary manager and function correctly for any extended length of time. In order to provide a continuous group communication mechanism, secondary managers must employ a scheme to detect the failure of the primary manager and select a new primary manager from among themselves. One possible scheme is the **death-will** scheme proposed by Ravindran [14]. In this case we assume that the underlying Kernel supports facilities for a process to create **aliases** that may reside in different address spaces (in the same or different machine) to perform functions related to failure detection and notification on behalf of their creator. Another scheme is that if the prober method is used by the primary manager to detect secondary manager failures, the lack of probes for extended period of time will indicate primary manager failure. However this scheme requires each secondary manager to create a **timer**. Another possible scheme is that each secondary manager may create a vulture process to look for the failure of the primary manager. Since the underlying system on which the prototype of the proposed mechanism is built supports such abstraction, this scheme has been implemented.

Every secondary manager is a potential candidate to become the next primary manager due to the fact that each of them has the same global view of the group and each of them has the capability of detecting primary manager's failure. The scheme to select a new primary manager must deal with several issues which may arise. For example, there may be inconsistencies due to two or more secondary managers attempting to become the new primary manager. Failures may even occur during the selection of the new primary manager itself.

Therefore the scheme must guarantee that when the selection is over, the group must be left with only one primary manager and all the secondary managers must know the identity of the new primary manager. The following section gives an overview of several possible selection schemes and in Section 3.4.1, a finite state model of the selection scheme used in the proposed group communication mechanism is presented.

3.4 New Primary Manager Selection Scheme : An Overview

Let us first examine some of the existing selection schemes which include token passing [3] and election [10]. The token passing scheme is suitable in an environment where the leadership is rotated among the members even when there are no failures, as in Chang's [3] reliable broadcast protocol implementation. The election scheme is suitable when a new leader is selected only if the old leader has failed. In the election scheme, each potential candidate does not have any knowledge about other candidates and depends on random timeouts before proclaiming itself as the new leader. Such a scheme is used in electing a leader in TEMPO, a distributed clock synchronizer running on UNIX 4.3 BSD systems [10]. To elect a primary manager we propose a scheme called the succession list scheme which is simpler than the election scheme. In this scheme each potential candidate has information about the other candidates. Normally this information is in the form of an ordered list. For example, the list may be ordered in increasing value of the candidate's pids or ordered by the time at which the candidates were created. All the candidates agree to select the first (or the last) candidate in the list as the successor when the leader fails.

In our group communication mechanism each secondary manager has it's manager member list ordered by the time they joined the group. Therefore the pid of the first secondary manager to join the group will be first after the

primary manager's pid in the manager member list, and the pid of the last secondary manager to join will be last in the list. In case of primary manager failure, the first operational secondary manager in the list will become the next primary manager. Based on this, one could propose a very simple succession list scheme where the younger secondary managers will wait until the oldest secondary manager inform them about the new leadership without running an agreement protocol among themselves. However, this will not work under certain circumstances. Suppose the oldest secondary manager also fails immediately after the primary manager has failed, the secondary manager's failure will not be notified to any of the operational secondary managers. In this case, if the younger secondary managers just wait to hear from the oldest secondary manager without probing it, then they may wait forever. Therefore it is necessary to run an agreement protocol among all the operational secondary managers before a new primary manager is selected. In the next section we use a finite state model to explain the details of our selection scheme.

3.4.1 Succession List Selection Scheme : A Finite State Model

During their lifetime, secondary managers can be in one of a finite number of states. Transition from one state to another is caused by the arrival of a message. A state transition may cause a secondary manager to transmit a message which triggers subsequent transitions in other secondary managers. It is important to clarify that in explaining this scheme we focus only on the state of one secondary manager, say sm(i), and not on the state of the entire distributed program. Figure 3.4 represents the state diagram for a secondary manager sm(i). Circles represent states, arrows represent transitions. A transition occurs upon the arrival of a message or the occurrence of an event. The event which causes the transition is shown on the upper part of the label associated with the arrow.

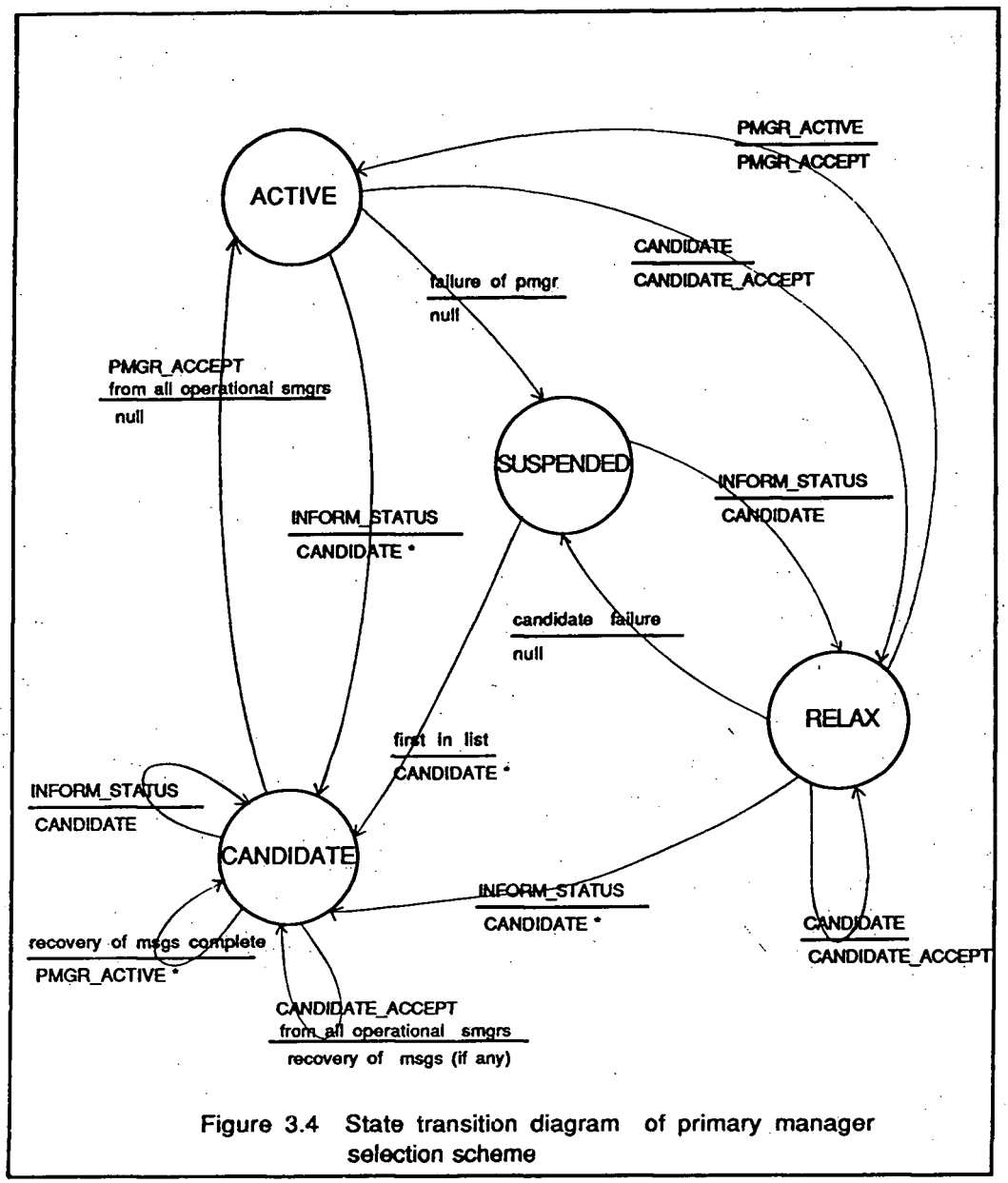


Figure 3.4 State transition diagram of primary manager selection scheme

The lower part of the label shows the message that is sent at the time of the transition. An asterisk by a message indicates, that it is a broadcast (i.e., the message is sent to all the secondary managers). A null label indicates that no message is sent or received.

3.4.1.1 Description of the States

active :

Normally the primary and the secondary managers of a group will be in the

active state.

suspended :

This is a transition state which is reached when secondary manager sm(i) learns about the failure of the primary manager. At this state, sm(i) checks to see if there are secondary managers in front of it in the manager member list. If so, it sends a probe message INFORM_STATUS to sm(j), the first among the secondary managers in the list ahead of sm(i). If the underlying system informs sm(i) that sm(j) is not operational, sm(i) probes the next secondary manager down the list.

However if sm(j) replies with the message CANDIDATE to the probe message, then sm(i) enters the **relax** state. On the other hand if sm(i) finds out that it is the first operational secondary manager in the manager member list, then it broadcasts the message CANDIDATE to all the secondary managers and enters the **candidate** state.

relax :

We have already seen how a secondary manager can reach this state from the **suspended** state. A secondary manager can also reach this state from the **active** state. For example if there is a delay in learning about the primary manager's failure, sm(i) will still be in **active** state. If it now receives the message CANDIDATE from sm(j) which is infront of sm(i) in the list, then sm(i) will enter the **relax** state after sending the reply message ACCEPT_CANDIDATE to sm(j). On entering the **relax** state, sm(i) notes down the candidate's identifier in it's *last_candidate* field. Also it informs it's vulture process to detect the failure of the primary manager candidate.

While sm(i) is in the **relax** state, the primary manager candidate may fail. When sm(i) learns about this failure from it's vulture it returns to the **suspended** state. However if there is a delay in learning about this failure, it may either receive the message INFORM_STATUS or CANDIDATE from some other primary managers. For example, if sm(i) is the next operational secondary manager in the list after the failed candidate, other secondary managers will probe it with INFORM_STATUS. On the other hand, the CANDIDATE message may be received from a secondary manager sm(k) which is the first operational secondary manager in the manager member list. Sm(i) in the **relax** state may also receive the CANDIDATE message in other circumstances. For instance, it may have entered the **relax** state from the **suspended** state because secondary manager sm(j) ahead of sm(i) in the manager member list has replied with the message CANDIDATE to sm(i)'s INFORM_STATUS probe. If this has happened before sm(j) broadcasts the message CANDIDATE, then sm(i) will receive the message CANDIDATE from sm(j) one more time.

If sm(i) receives the CANDIDATE message while in the **relax** state, it will compare the pids of the last_candidate and the sender of the message. If they are different, then a new candidate has initiated the election. Sm(i) instructs it's vulture to look for the failure of the new candidate, notes down the new candidate's pid in it's last_candidate field, and replies to the new candidate with the ACCEPT_CANDIDATE message. However if the pids are the same, then sm(i) is already monitoring the right candidate and therefore it simply replies with the ACCEPT_CANDIDATE message.

candidate :

This is the state when a secondary manager declares it's intention to contend for the primary manager position. As explained in the **suspended** state, the first

operational secondary manager in the manager member list will reach this state after broadcasting the CANDIDATE message to all the secondary managers. Secondary manager sm(i) can also reach this state from the **active** state. For example, if there is a delay in learning about the failure of the primary manager, sm(i) will still be in the **active** state. While in the **active** state, if sm(i) receives an INFORM_STATUS message from secondary manager sm(k) which is behind it in the manager member list, sm(i) will reply to sm(k) with the CANDIDATE message. It will then broadcast the CANDIDATE message to all the secondary managers and enter the **candidate** state. Secondary manager sm(i) can also enter the **candidate** state from the **relax** state as explained under **relax**.

If in the **candidate** state sm(i) receives an INFORM_STATUS message, it will reply with the message CANDIDATE. After validating that all the secondary managers have received it's CANDIDATE message and have reached the **relax** state (i.e., after receiving CANDIDATE_ACCEPT message from all the operational secondary managers), the primary manager candidate will finish any incomplete OGSEND or group view message transmissions initiated by the failed primary manager. The new primary manager will then broadcast the message PMGR_ACTIVE to all the secondary managers. On receiving that message, the secondary managers reply with the message PMGR_ACCEPT, update the manager member list, enter the **active** state and resume normal operations. After the new primary manager has received the PMGR_ACCEPT message from all the operational secondary managers, it creates a prober process and enters the **active** state to resume normal operation.

3.5 Network Partition

When the network partitions, the system divides into two or more subgroups of hosts; all but one subgroup will be left without a primary manager. In our scheme, secondary managers within a subgroup will select a primary manager and continue to function. The proposed mechanism assures that the communication within these subgroups will be ordered and atomic. The difficult problem is what happens when the partitions merge again.

For example, consider a group with four secondary managers as shown in Figure 3.5(a). The secondary managers and their hosts are identified by the numbers 2 through 5 with primary manager M of the group assigned the number 1. Assume that the multicast address of this group is X. The group view maintained by the primary and all the secondary managers is then [multicast address X; primary manager 1; manager member list 1,2,3,4,5].

Consider now a network partition which separates hosts 4 and 5 from the rest of the group. As a result, there will be two subgroups: subgroup A consisting of the primary manager and the secondary managers 2 and 3, subgroup B with secondary managers 4 and 5. Since the underlying system in

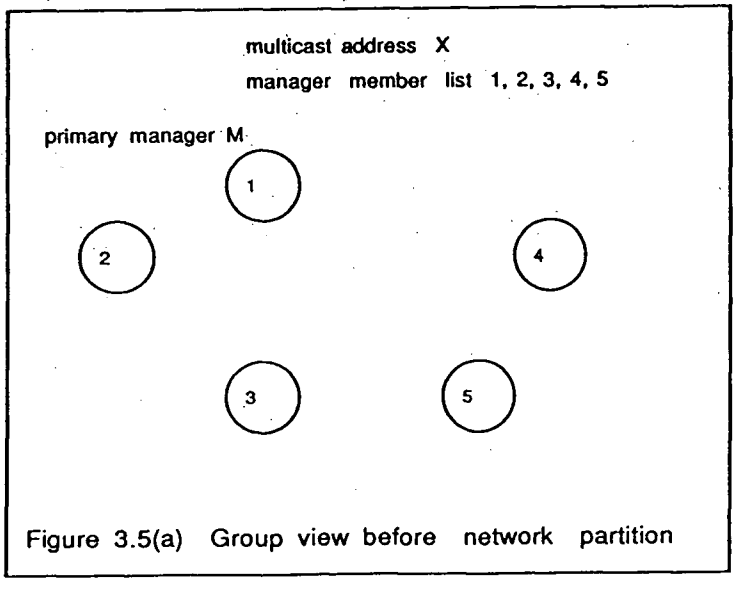


Figure 3.5(a) Group view before network partition

the primary manager's host cannot distinguish between network partition and host failure, it will inform the **prober** that secondary managers 4 and 5 have failed. This results in the removal of secondary managers 4 and 5 from the manager member list maintained by the primary manager M and secondary managers 2 and 3. Thus, the group view in subgroup A will be [multicast address X; primary manager 1; manager member list 1,2,3]. On the other hand, secondary managers in subgroup B will be notified by their **vultures** that the primary manager has failed and secondary managers 4 and 5 will select a primary manager among themselves. Assume that secondary manager 4 has been selected as the primary manager M' of subgroup B. The group view in subgroup B is then [multicast address X; primary manager 4; manager member list 4,5]. This is illustrated in Figure 3.5(b). As long as the partitioned network does not remerge, communication within these subgroups will have the same reliable properties discussed in Chapter Two.

After some period of time, the partitioned network may remerge. If the subgroups do not merge to form a single group with one primary manager, reliable communication cannot be guaranteed. Consider a UGSEND activity

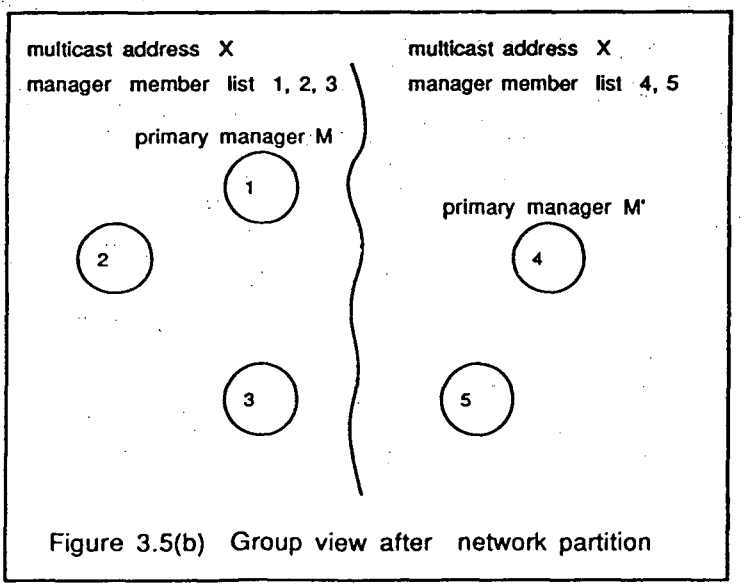


Figure 3.5(b) Group view after network partition

initiated by secondary manager 5 in subgroup B after the network has remerged. First, this secondary manager multicasts the message to the address X. Then it will resend the message to those in subgroup B that failed to receive the message the first time around. This will guarantee that the UGSEND message will be received by all the members in subgroup B. However, the UGSEND message sent to the multicast address X will also be received by the primary and the secondary managers in subgroup A, as they are also listening on the same multicast address.

The other problem after merging is due to the fact that there are multiple primary managers listening on the same multicast address thus confusing those wishing to interact with the group as illustrated in Figure 3.5(c). It is therefore necessary that the subgroups formed during network partition should merge to form a single group with one primary manager. Also the primary as well as all the secondary managers of this single group should have the same group view after the merge.

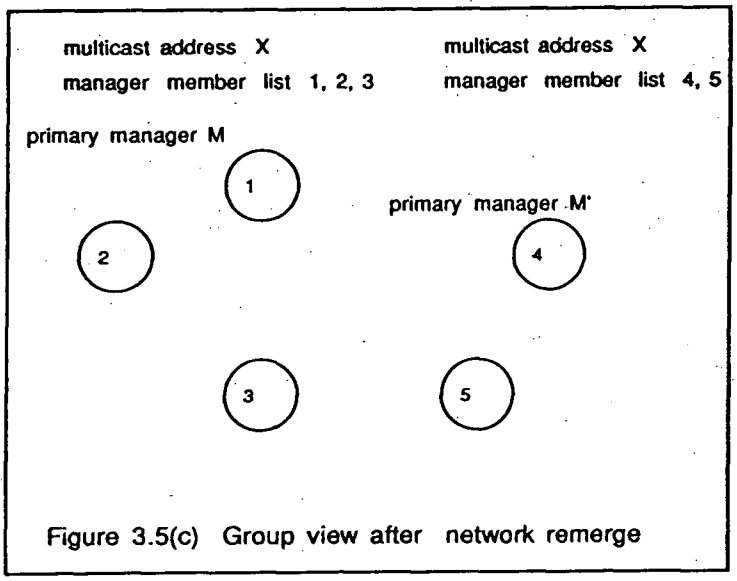


Figure 3.5(c) Group view after network remerge

In the follwing section we describe how our scheme handles the messages initiated by the primary or the secondary managers of one subgroup but received by the managers belonging to another subgroup. Section 3.5.2 describes how to merge multiple subgroups after the partitioned network has remerged.

3.5.1 Discarding Messages from Different Subgroups

After the network has partitioned, the communication within the subgroups will have the reliable properties discussed in Chapter Two. After the network has remerged, these subgroups will be merged together to form a single group. However, in the mean time, message transmission initiated from one subgroup may be received by the managers belonging to a different subgroup. This kind of reception should either be avoided or if it cannot be avoided, the received messages should be detected and discarded.

A simple scheme to attempt to avoid receiving these messages would be for each subgroup which selects a new primary manager to choose a new multicast address. When the primary manager is selected, it chooses a new multicast address and informs it's secondary managers about it. For example, in Figure 3.5(b), subgroup B which selects a new primary manager chooses a new multicast address X'. Thus the group view maintained by the primary manager M' and it's secondary manager 5 will be [multicast address X'; primary manager 4; manager member list 4,5]. If the network remerges, then the message transmission initiated by the primary or the secondary managers in subgroup B will never be received by the managers in subgroup A as they will be listening on multicast address X. One disadvantage of this scheme is that a new multicast address has to be selected whenever a new primary manager is chosen. This may happen even when the network is not partitioned (i.e., just a primary manager failure). Also, there is no guarantee that independently chosen multicast

addresses on different partitions are distinct. This will cause problems when the partitions remerge.

In the proposed mechanism, we use another scheme where the messages received by the primary or secondary managers of a subgroup will be discarded if they were not initiated from within their subgroup. In this scheme when a manager transmits a message, it specifies it's primary manager's identifier in the message header. Whenever a manager receives a OGSEND or UGSEND message, it compares it's primary manager's pid against the primary manager's pid specified in the message header. If they are different, the message will be discarded. For example, if secondary manager 5 transmits a message after the merger of the partitioned network (but before the merger of the subgroups) to the multicast address X, even if the primary or secondary manager of subgroup A receives this message, it will be discarded as the primary manager M of subgroup A will be different from the primary manager M' specified in the message header†.

3.5.2 Merging Subgroups

When a partitioned network merges, the subgroups formed due to this partition have to be merged. The primary managers of different subgroups listening on the same multicast address must detect that more than one primary manager exists for the same group and reach an agreement as to which should become the leader.

3.5.2.1 Detection of Subgroups

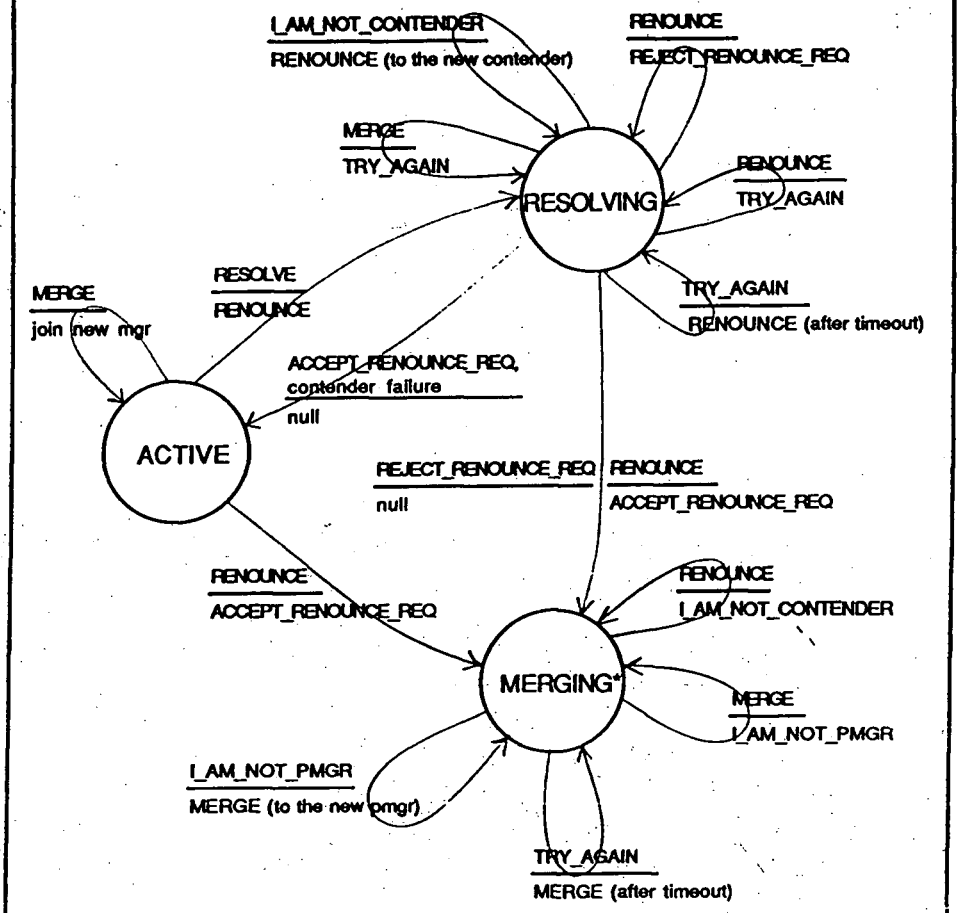
---

† Here, we are assuming either the hardware address is used as the identifier or that new primary manager will be chosen only from those secondary managers that already exist at the time of network partition. This is the most likely scenario and will guarantee the primary managers in different subgroups will have different identifiers.

In the proposed mechanism, the prober process periodically broadcasts a RESOLVE message to the group's multicast address. This message contains the primary manager's pid and are discarded by the secondary managers. Only primary managers respond to the RESOLVE message. To explain this scheme, consider Figure 3.5(a). When the network is not partitioned, the prober of primary manager M will be broadcasting the RESOLVE message which will be discarded by secondary managers 2 through 5. After network partition (Figure 3.5(b)) there are two subgroups; one with primary manager M and the other with primary manager M'. The RESOLVE message broadcast by the probers of primary managers M and M' will be discarded by the secondary managers 2 and 3 in subgroup A and secondary manager 5 in subgroup B. After the network remerges, the multicast address of the two subgroups will still be X. Thus, the RESOLVE message of primary manager M may be received by primary manager M' and vice versa. When a primary manager receives a RESOLVE message, it knows that the network has been partitioned and remerged. It also knows the identity of the other primary manager.

3.5.2.2 Resolving the Leadership

Once a primary manager detects that there exists other primary managers with the same multicast address, all except one of them has to renounce the leadership. We will make use of the finite state model shown in Figure 3.6 to explain the scheme used to resolve the leadership. The scheme works with merging a pair of primary managers at a time. In explaining this scheme we focus only on the state of a primary manager, say pm(i). In the figure, circles represent states, arrows represent transitions. A transition occurs upon the arrival of a message or the occurence of an event. The event which causes the transition is shown on the upper part of the label associated with the arrow.



* as soon as a primary manager enters the merging state, it initiates procedure to merge with the chosen primary manager and becomes a secondary manager when the merge is complete (see Section 3.5.2.2)

Figure 3.6 State transition diagram of primary managers resolving leadership upon network remergence

The lower part of the label shows the message that is sent at the time of the transition. A null label indicates that no message is sent or received.

Description of the States

active :

Normally pm(i) will be in the **active** state. If it receives a RESOLVE message

in this state from another primary manager, say pm(j), it sends a RENOUNCE message to pm(j), notes down the pid of pm(j) in it's *contender_pid* field and enters the **resolving** state. The RENOUNCE message also includes the number of secondary managers in the manager member list of pm(i) and indicates the intention of the sender to contend for leadership.

Sometimes it is possible that pm(i) in the **active** state may receive a RENOUNCE message from pm(j) before it receives the RESOLVE message. When a primary manager receives a RENOUNCE message, whether it gives up it's leadership or not depends on the number of secondary managers in it's manager member list. In the proposed scheme, the primary manager with the most number of secondary managers will become the leader. If both contenders have equal number of secondary managers, then the one with the larger pid will assume the leadership. Thus, if pm(i) has less number of secondary managers (or equal number of secondary managers but smaller pid), then pm(i) will reply with an ACCEPT_RENOUNCE_REQ message and enters the **merging** state. However, if pm(i) has more number of secondary managers (or equal number of secondary managers but larger pid), then it will reply with a REJECT_RENOUNCE_REQ message and remains in the **active** state.

While in the **active** state, pm(i) may receive a MERGE request from another manager. Pm(i) simply joins the merging manager in it's group (see **merging** state).

resolving :

We have seen under **active** how a primary manager enters the **resolving** state from the **active** state after sending a RENOUNCE message to pm(j). If it receives an ACCEPT_RENOUNCE_REQ message from pm(j) in response to it's RENOUNCE message, pm(i) will return to the **active** state. Pm(i) also returns to

the **active** state if pm(j) fails. On the other hand if pm(i) receives a REJECT_RENOUNCE_REQ message from pm(j), it enters the **merging** state.

Pm(i) which has entered the **resolving** state after sending a RENOUNCE message to pm(j) may receive a RENOUNCE message. This message may be from the primary manager pm(j)† or from a third primary manager, say pm(k). Let us first consider the case of receiving this message from pm(j). If pm(i) determines that pm(j) is the eligible contender, then it replies with an ACCEPT_RENOUNCE_REQ message and enters the **merging** state. However, if pm(i) finds that pm(j) is not the eligible contender then it replies with a REJECT_RENOUNCE_REQ message and returns to the **active** state.

If the RENOUNCE message was sent by pm(k), pm(i) will detect this by comparing it's *contender_pid* with the pid of the process sending the RENOUNCE message. This may happen if the network partition had divided the group into more than two subgroups which subsequently remerged. In this case pm(i) replies with a TRY_AGAIN message meaning that pm(i) is busy resolving the leadership with another contender and therefore pm(k) should wait and try again later.

merging :

The primary manager pm(i) enters this state from the **active** state or from the **resolving** state as explained earlier. While in the **merging** state, pm(i) sends a MERGE request to it's new primary manager who will accept pm(i) as a new secondary manager in it's group and exchange with it new group view information (same procedure as the case of a new secondary manager joining the group - see Section 3.1). It is possible that pm(i) may receive a RENOUNCE

---

† It may appear as if there will be a communication dead lock, but this can be overcome as explained in Section 4.6.

message while in the **merging** state. In such an event, it replies with a I_AM_NOT_CONTENDER message which includes the pid of the new primary manager. The sender of the RENOUNCE message will then try to resolve the leadership with it's new contender.

Also, it is possible that pm(i) may receive a MERGE request while in the **merging** state for which it replies with a I_AM_NOT_PMGR message. This message includes the pid of the new primary manager so that the merging manager may request to merge with the new primary manager.

When pm(i) sends a MERGE request to a primary manager which is in the **resolving** state, the latter replies with a TRY_AGAIN message and pm(i) will retransmit the MERGE request after some specified time period. When the new primary manager receives a MERGE request from a manager it accepts the merging manager as a new secondary manager in it's group, i.e., it transfers the group view information to the merging manager, updates it's manager member list with the pid of the merging manager and informs all the secondary managers of the group to update their lists as well.

3.6 Chapter Summary

The group communication mechanism requires some form of coordination to realize the reliable properties. In the proposed mechanism each group has a primary manager to coordinate the group management and communication activities. In order to ensure survivability in case of primary manager failure, the primary manager is replicated in all the member sites and a new primary manager is selected from among these secondary managers. The secondary managers do not take part in any group management activities, but may take part in communication activities when ordering is not a requirement. The prober

process executing in the primary manager host detects any secondary manager failure and notifies the primary manager. The primary manager must finish up any incomplete message transmission initiated by the failed secondary manager. A vulture process executing in each of the secondary manager hosts detects primary manager failure and notifies it's secondary manager. The secondary managers then select a new primary manager using a succession list selection scheme. Network partition may result in subgroups of sites with the same multicast address. The proposed mechanism ensures that communication within these subgroups will continue to exibit the reliable properties. When the partitioned networks remerge, the proposed mechanism detects the different subgroups and merges them to form a single group.

Chapter Four
Implementation Details of the Proposed Group Communication Mechanism

This Chapter describes the implementation details of the proposed group communication mechanism and it's performance. One has basically two choices in implementing the proposed mechanism; either to implement it as part of the kernel of a distributed system or to implement it on top of an existing well tested kernel. Because of time constraint, and since the primary object is to test the feasibility of the proposed mechanism rather than it's performance, we have chosen the second approach. The proposed mechanism is built on top of the V Distributed System running on a cluster of workstations in our Distributed System Research Laboratory.

In implementing the proposed mechanism, three major issues have to be dealt with; group management, group communication, and failure detection and recovery procedures. Group management addresses such issues as group creation and processes joining or leaving a group. Group communication deals with the issue of transferring a message from a source to all the members of the group with the reliable properties discussed in Chapter Two. Failure detection and recovery is essential for the proposed mechanism to provide continuous service despite host failures and network partitioning.

The proposed mechanism is structured as a set of cooperating processes; the primary and secondary managers are examples of such processes. In addition to these management processes, there are **worker** processes to help the manager processes to achieve the desired reliable properties. The **vulture** and **prober** are two examples of the worker processes.

This Chapter describes the implementation details of the above aspects and is divided into the following sections. Section 4.1 describes the implementation details of group management. In order to manage a group, it's manager processes maintain some group management information. The manager member list in which the pids of the primary as well as the secondary managers for the group are maintained is an important part of the group management information. Section 4.2 describes the organization of the manager member list. In Section 4.3, implementation details of the group send primitives **ugsend**() and **ogsend**() are described. Section 4.4 describes the details of the worker processes. Failure detection and recovery procedures in the event of host failures and network partitioning are explained in Sections 4.5 and 4.6 respectively. Performance evaluation of the group send primitives provided by the proposed mechanism is given in Section 4.7, and Section 4.8 concludes the Chapter.

4.1 Group Management

The proposed group communication mechanism provides facilities to transfer a message from a source to a set of processes called a group. Thus, in addition to communication, the mechanism should provide facilites for the application processes to create, join, and to leave a group. In order to provide these functionalities, the hosts which support the proposed group communication mechanism run a process called the **group server**. Processes which wish to create or to join a group invoke the group management stub routines which in turn send appropriate requests to the group server.

The following section describes the **creategroup**() routine which is invoked by a process wishing to create a group. Once a group is created, the group must be associated with a logical name (i.e., group name) so that processes will be able to interact with the group using this logical name. Section 4.1.2

describes the detail of registering a group with the name service. Sections 4.1.3 and 4.1.4 describe the **joingroup**() and **leavegroup**() routines invoked by processes wishing to join or leave a group respectively.

4.1.1 Creating a Group

A new group is dynamically created when a process invokes the **creategroup()** routine. This is a stub routine which sends a CREATE_GROUP request along with the invoker's (initial member) pid to the group server. The group server creates a primary manager for the group and sends this request to it. The primary manager simply adds the initial member's pid to the local group member list. The group server then returns the primary manager's pid to the invoker of the **creategroup** routine.

4.1.2 Registering a Group

In order to make the primary manager available to the processes wishing to interact with the group, it's pid should be associated with a logical id. Since the prototype of our mechanism is built on top of the V Kernel, we make use of the name service facility provided by the underlying system. In the V Kernel, when a process wants to associate it's pid with a logical id, it invokes **SetPid**(logical_id, pid, scope). If the specified scope is LOCAL, then the pid is registered locally so that only processes executing in the same host can obtain the pid from the name service. On the other hand if the specified scope is ANY, then the pid is registered globally so that processes executing in any host in the network will be able to obtain this pid from the name service.

When a process wants to find out the pid associated with the logical id, it invokes **GetPid**(logical_id, scope) which returns the pid of the process registered in the name service using the **SetPid** routine. If the specified scope is

LOCAL, then the name service returns a pid of a process locally registered to the invoker's host. However, if the scope is ANY, then the name service first looks for a locally registered process. If one is not found, it broadcasts a request to other hosts in the network requesting them to send it the pid associated with the logical id, if there is any.

In order to associate the primary manager of the group with a group name, the creator of the group invokes **registergroup**(groupname, pmgr-pid, type). If the specified type is LOCAL, then only processes residing on the same host as the primary manager will be able to obtain the primary manager's pid from the name service. However, if the type is GLOBAL, then processes from any hosts in the network will be able to obtain the primary manager's pid from the name service. **Registergroup** is a stub routine which sends a REGISTER_GROUP request to the primary manager whose pid is specified by pmgr-pid. The primary manager invokes the SetPid routine to register the group name in the name service. Normally the primary manager will register with type GLOBAL unless the group is meant to be a local group only. As described in the next section, secondary managers also register with the **registergroup** routine, but the type is always LOCAL.

4.1.3 Joining a Group

Processes wishing to join a group first find the group_id associated with the group name and then invokes **joingroup**(group_id). The group_id returned by the name service may be the pid of the primary or secondary manager for that group, depending on the location of the joining process as explained below.

Let us consider the case where a member joins the group from a host where neither the primary manager nor a secondary manager resides. In this

case, the group_id returned by the name service will be the primary manager's pid (assuming that the group has been registered as GLOBAL). After obtaining the group_id, the joining member invokes **joingroup** routine to join the group. This is a stub routine which sends a JOIN_GROUP request to the groupserver along with the group_id and the joining process' pid. The group server creates a secondary manager for this group in the joining process' host and informs the primary manager (group_id) about the new seconadry manager. The primary manager then transfers it's group management information to the new secondary manager which includes the manager member list and the group name. The primary manager then updates it's manager member list with the new secondary manager's pid and informs all the other secondary managers of the group to update their lists as well.

When a secondary manager receives the group management information, it associates it's pid with the received group name and registers in the name service with LOCAL scope. Thus, later on, when a process residing in the secondary manager's host requests the name service for the pid associated with the group name, it will obtain the secondary manager's pid.

After transferring the group management information to the newly created secondary manager, the primary manager notifies the group server. The group server then sends a JOIN_MEMBER request to the secondary manager along with the joining process' pid. The new secondary manager simply adds this pid to it's local group member list.

When a member joins the group from a host where a secondary manager for this group already exists, then the group_id specified in the **joingroup** routine will be the local secondary manager's pid. Thus, it is not necessary to create a new secondary manager in the joining process' host. The group server

simply sends a JOIN_MEMBER request to the local secondary manager along with the joining process' pid. The local secondary manager adds the new member to it's local group member list.

4.1.4 Leaving a Group

A process wishing to leave a group invokes the **leavegroup**(group_id) routine. This is a stub routine which sends a LEAVE_GROUP request to the process specified by the group_id. This group_id corresponds to the pid of either the primary or a secondary manager for the group depending on the location of the exiting process as explained in Section 4.1.3. When the primary or a secondary manager receives the LEAVE_GROUP request along with the exiting process' pid, it simply deletes the member's pid from it's local group member list. If the exiting process is the only member in the secondary manager's local group member list, then the secondary manager sends a NO_LOCAL_MEMBERS message to the primary manager which deletes this secondary manager from it's manager member list and informs all other secondary managers of the group to do the same. Finally, the primary manager sends a COMMIT_SUICIDE message to the memberless secondary manager which deletes it's pid from the name service and ceases execution. However, if the leaving process is the only member in the primary manager's local group member list, it has to make sure that it's manager member list is empty before ceasing execution. If the manager member list is not empty, then the primary manager simply deletes the leaving member's pid from it's local group member list and continues to function. When the primary manager ceases execution, the group becomes nonexistent.

4.2 Organization of the Manager Member List

The group management information transferred from the primary manager to a newly created secondary manager includes the pids of the primary as well as all the secondary managers. These pids are maintained in a table called the **manager identifier table.** To find a particular manager's identifier, one can use that manager's **manager index** to index into the manager identifier table. The manager indices are assigned by the primary manager. These indices are also part of the group management information transferred to a newly created secondary manager.

Initially the primary manager is assigned manager index 0. The first secondary manager to join the group will be assigned manager index 1 and the following secondary manager to join will be assigned manager index 2 and so on. Suppose the secondary manager with manager index 1 fails, the primary manager makes manager index 1 invalid, and informs the other secondary managers to do the same. Later, manager index 1 may be assigned to a new secondary manager by the primary manager. Manager indices are maintained in a **manager index list** in such a way that the primary manager's index will

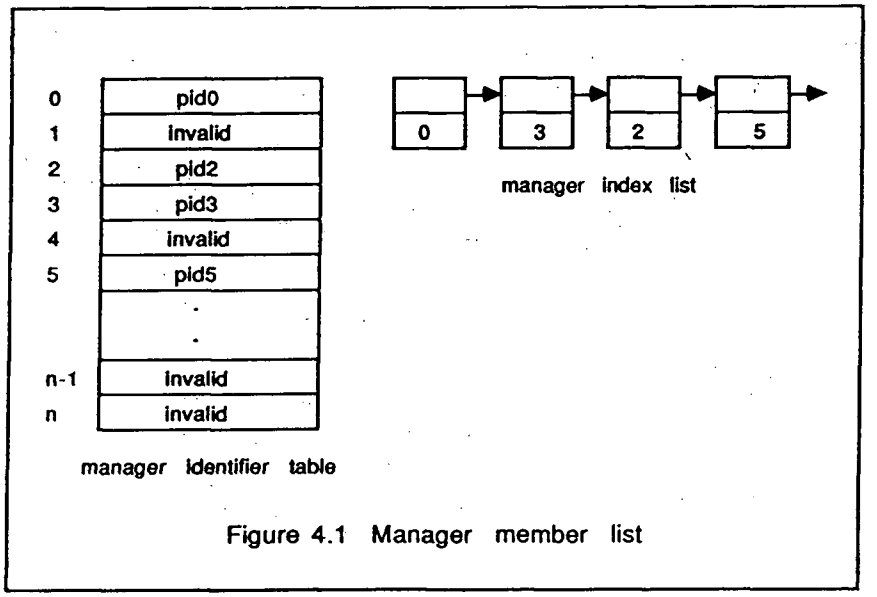


Figure 4.1 Manager member list

always be first in the list and the index of the last secondary manager to join will be at the end of the list. This ordering is essential for the selection scheme to choose a primary manager in case the old primary manager fails as described in Section 3.4.1. The manager indentifier table and the manager index list are together called the **manager member list**, as illustrated in Figure 4.1.

4.3 Group Communication

This section describes the implementation details of the **ogsend** and **ugsend** primitives used by the applications to transfer a message from a source to the members of a group.

4.3.1 Ugsend Implementation

Processes send UGSEND type messages by invoking **ugsend**(msg, group_id, msgtype) to the members of the group whose group name is associated with the specified group_id. If the IMMEDIATE_REPLY bit is set in msgtype, then the sender may be unblocked by the group communication mechanism before the message is delivered to the members of the group. Otherwise the sender will be unblocked only after the members have received and acknowledged the message. The specified group_id may be either the primary or a secondary manager's pid depending on the sender's location. **Ugsend** is a stub routine which sends a UGSEND_MSG request embedded with the message to the specified group_id. The primary or the secondary manager which receives this request has two ways to transmit the message to other managers depending on the underlying network architecture. If it supports only unicast facility, then the messages are sent on a one-to-one basis to the individual managers. However, if the underlying network also supports broadcast facility then the messages can be first multicast to the managers in a datagram

fashion and later resent on a one-to-one basis to those who fail to receive the message the first time around. Since the underlying network architecture of our environment supports the broadcast facility, we use the second scheme. Thus, when the primary or the secondary manager receives UGSEND_MSG, it first multicast the message to the rest of the managers. It then waits for a specified period of time to receive acknowledgements from the recipients. If acknowledgements are not received from some managers at the expiration of the time interval, the message is resent to them using one-to-one IPC's.

On the receiving side, the recipient managers can send acknowledgements back to the sending manager immediately after receiving the message or only after delivering the message to their local members depending on whether the IMMEDIATE_REPLY_BIT is set or not in the opcode specified in the message header.

4.3.2 Ogsend Implementation

Messages sent from different sources by invoking the **ogsend** routine will be delivered in the same order to all the members of the group. We have seen in Chapter Two that this ordering can be easily achieved by funneling the OGSEND messages through a single process. In the proposed mechanism the primary manager acts as the funnel process. Processes wanting to send OGSEND messages invoke **ogsend**(msg, group_id, msgtype). The specified group_id may be the primary or secondary manager's pid depending on the sender's location. Therefore, the stub routine **ogsend** has to first find the primary manager's pid and then send the message to it for transmission. Thus, it sends a GET_PMGR_PID request to the process specified by group_id. Since all the managers of the group know the pid of the primary manager, this information is available whatever the specified group_id. Once the primary manager's pid is

obtained, the stub routine sends the OGSEND message to the primary manager which transmits this message to all the secondary managers in a similar fashion explained for UGSEND transmission.

4.3.3 Detection of Duplicates

In the UGSEND or OGSEND message transmission, if the the message is transmitted to the rest of the managers using one-to-one IPC, then the recipients will not receive duplicate messages. However if the message is multicast first in datagram fashion and later resent on a one-to-one basis, then some recipients may receive duplicate messages as explained in Section 3.2. This section describes the implementation details of the duplicate detection scheme. We will describe the scheme from the UGSEND message transmission point of view. Similar technique is used in OGSEND message transmission.

The primary and the secondary managers maintain an integer variable **ussno** which is the sequence number of the next UGSEND message. They also maintain **receive buffers** where the last UGSEND message sent by other managers can be stored. Receive buffers corresponding to a particular manager is indexed by it's manager index. Each receive buffer has two fields: a message buffer to store the last UGSEND message and an integer variable, **ursno**, to keep track of the transaction identifier of the next incoming UGSEND message sent by the manager whose manager index indexes into this receive buffer as illustrated in Figure 4.2. When the primary or secondary manager transmits a UGSEND message, the message header contains the sender's manager index, **ussno** and the pid of the primary manager for that group. When an UGSEND message is received by a manager, it first checks the primary manager pid specified in the message header against it's primary manager's pid. If it is different, then this message must have been transmitted from a manager

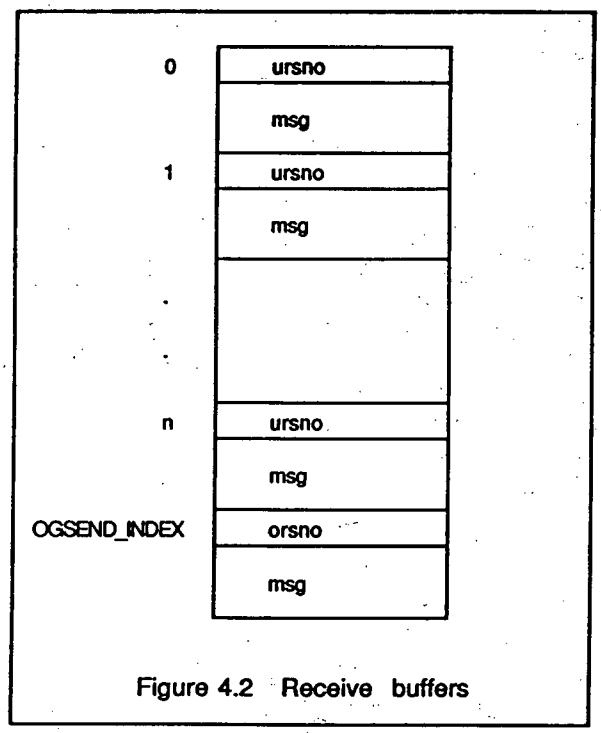


Figure 4.2 Receive buffers

belonging to another group having the same multicast address. This may have happened as a result of the underlying network partitioning and remerging. However, if the message transmission was initiated within the same group, then the receiving manager compares the **ussno** specified in the message header against the **ursno** corresponding to the receive buffer indexed by the manager index. If this is the expected message from the sending manager it will be accepted else it will be discarded. Once the primary or secondary manager accepts a message, it can reply to the sending manager immediately if the IMMEDIATE_REPLY_BIT is set in the opcode of the message header. Otherwise it replies only after the message has been delivered to and acknowledged by it's local members.

4.4 Worker Processes

The primary and secondary managers are responsible for activities pertaining to the group management and the group communication. In order to improve concurrency which normally improves the performance of a distributed program, the manager processes employ some worker processes to carry out some of their tasks. **Vultures** and **probers** described in Section 3.3.1 are examples of such processes. In addition to these processes, managers employ **courier** processes to carry out the message transmission activities, **aide** processes to chose a new primary manager in case of primary manager failure, **resolver** processes to resolve the leadership in situations such as when a partitioned network remerges. This section describes the implementation details of these worker processes. The worker processes share the same address space as their managers and thus have read access to the group management information mainatined by their managers.

4.4.1 Courier

Couriers are responsible for carrying out the UGSEND and OGSEND message transmission activities on behalf of their managers. A high level description of the courier process is illustrated in Figure 4.3. Couriers are created by the primary and secondary managers when they are initialized. Since the

```
Type  : courier
Task  : sending a group message

FOREVER DO
    Begin
      ReceiveSpecific (from primary manager)
      Send (to all managers) { group send if broad-
                               cast available}
      Reply (to primary manager)
    End
```

Figure 4.3 Courier process

primary as well as the secondary managers can take part in the UGSEND activity, each manager has a UGSEND courier. Also, both primary and secondary managers have a local courier to help them deliver the messages to their local members. In additon to UGSEND activity, the primary manager is responsible for transmitting OGSEND messages and group management messages. Thus, the primary manager has an additional OGSEND courier to assist in transmitting these type of messages.

When the primary or secondary manager receives a message for transmission, it first checks if the courier appropriate for handling the transmission is free. If the courier's status indicates that it is FREE, then the message is handed over to it for transmission to all the managers of the group. Once the message has been handed over to the courier, it's status is set to BUSY. After the message has been delivered to and acknowledged by all the operational managers for the group, the courier notifies it's manager which then sets the courier's status to FREE. If the primary or secondary manager receives a message for transmission while the appropriate courier is busy handling the previous message, then the new message is queued first-in-first-out in the courier's message queue. After it completes the message transmission, the courier picks up the next message from it's message queue if it is nonempty.

4.4.2 Prober

The prober process is created by the primary manager to probe the secondary managers of the group to determine if they are still operational. The prober periodically (every 30 seconds in our implementation) sends a probe message ARE_YOU_ALIVE to all the secondary managers in the manager member list using one-to-one IPC. The operational secondary managers reply with a I_AM_ALIVE message to this probe. If a secondary manager has failed, then

the underlying system will notify the prober that it is trying to send a message to a nonexistent process. When the prober learns about this failure, it sends a SMGR_FAILED message to the primary manager along with the failed secondary manager's pid. In addittion to this probing, the prober is also responsible for multicasting a RESOLVE message periodically (every two minutes in our implementation). This message is necessary to detect whether there are any other groups listening on the same multicast address which may happen if the underlying network partitions and remerges as explained in Section 3.5.

4.4.3 Vulture

Each secondary manager for a group employs a vulture to detect the failure of the primary manager for that group. A high level description of the vulture process is shown in Figure 3.2. The vulture process makes use of the **ReceiveSpecific** primitive provided by the underlying V Kernel to detect the failure of the primary manager. Basically the vulture is simply receive blocked on the primary manager. It will be unblocked only if the primary manager sends a message to it or if the primary manager fails. In the latter case, the underlying system informs the vulture that it is trying to receive a message from a nonexistent process. When the vulture learns about the primary manager's failure, it sends a PMGR_FAILED message to it's secondary manager which then takes part in the selection of a new primary manager. After a new primary manager has been selected, the secondary managers inform their vultures to look for the failure of the newly selected primary manager.

4.5 Failure Detection and Recovery

This section describes the implementation details of failure detection and recovery procedures in case of secondary manager failure (4.5.1) and primary

manager failure (4.5.2).

4.5.1 Secondary Manager Failure

We have described in Section 4.4.2 how the prober detects the failure of a secondary manager. If a secondary manager fails in the middle of a UGSEND message transmission, some members may not receive the message. Thus, when the prober informs the primary manager that a particular secondary manager has failed, the primary manager sends a SEND_LAST_MSG request to all the other operational secondary managers. This message contains the failed secondary manager's index. The operational secondary managers which receive this request send the last UGSEND message received from the failed secondary manager. This message is stored in the receive buffer of each operational secondary manager indexed by the failed secondary's manager index. If the returned messages as well as the last UGSEND message received by the primary manager from the failed secondary manager have the same **ussno**, then the failed secondary manager has either successfully completed it's last UGSEND message transmission activity or no member has received it's last UGSEND message. Either of these outcomes assures *atomicity*. However, if there is a discrepancy among the **ussnos**, then the primary manager takes the message with the highest **ussno** and retransmits it to the secondary managers. Those secondary managers that have already received this message simply discard the duplicates as their **ursnos** will not match the **ussno** of the retransmitted message. However, those that have not received the message from the failed secondary manager receive this message and deliver it to their local members.

After finishing the incomplete UGSEND message transmission, the primary manager deletes the failed secondary managers's pid from it's manager identifier table and invalidates that secondary's manager index. This information is also

sent to all the operational secondary managers for them to update their lists.

4.5.2 Primary Manager Failure

A group cannot function without a primary manager. Thus, when the primary manager of a group fails, the secondary managers for that group must detect this failure and select a new primary manager from among themselves. We have described in Section 4.4.3, how the secondary managers detect the failure of the primary manager through their vultures, and in Section 3.5 about the selection scheme used to choose the new primary manager. In order to avoid possible communication deadlocks, each secondary manager participating in the election creates an **aide** process which takes part in the message transmission activities (pertaining to elction) on behalf of it's secondary manager. These aide processes destroy themselves once their task is complete. Once a new primary manager is chosen, it must finish any incomplete transmission of OGSEND message or messages initiated by the failed primary manager to inform the secondary managers of changes in the group view such as a secondary manager has joined the group or has failed.

When the secondary managers receive these messages, they store them in a fixed receive buffer indexed by a value called OGSEND_INDEX common to all managers. OGSEND_INDEX. This index does not change with primary managers. Thus, when a new primary manager is chosen, it sends the SEND_LAST_MSG request to all the operational secondary managers. This request contains the fixed OGSEND_INDEX instead of a manager index. The operational secondary managers then return the last message stored in their receive buffers indexed by OGSEND_INDEX. If the returned messages as well as the last message received by the new primary manager have the same **ossnos**, then this message has either been successfully delivered to all the secondary managers or to none

of them. However, if there is a discrepancy, then the primary manager takes the message with the highest **ossno** and resends it to the secondary managers in a similar fashion explained for secondary manager failure. Once this is completed, the new primary manager reregisters it's pid with the name service with type GLOBAL so that processes executing on other hosts will be able to obtain it's pid from the name service. The new primary manager then creates a prober and an OGSEND courier processes, sets it's **ussno** (to be assigned to the next OGSEND message) to the value of the last **ussno** plus one, and resumes normal operation.

4.6 Network Partition

We have seen in Section 3.4 how network partition creates subgroups with the same multicast address. This causes problems when the partitioned network remerges. We have seen in Section 3.5.1 how the primary managers detect that there are more than one primary manager listening on the same multicast address, and in Section 3.5.2 about the leadership resolution scheme used to resolve the leadership. In order to avoid possible communication deadlocks, each primary manager participating in the leadership resolution creates a **resolver** process which takes part in the message transmission activities (pertaining to leadership resolution) on behalf of it's primary manager.

The primary manager which gives up it's leadership changes it's type to SECONDARY and informs the secondary managers in it's manager member list about their new primary manager. Each of the secondary managers then change it's state to MERGING and send a MERGE request to the new primary manager. When a primary manager receives a MERGE request from a secondary manager, it transfers the group view information to the merging manager and adds it's pid to it's manager member list. Also, it informs the secondary

managers that are already in it's manager member list to update their lists as well. Once a secondary manager has merged, it changes it's state to ACTIVE and resumes normal operation.

4.7 Performance of the Group Send Primitives

We have done some preliminary measurements on the elapsed time for the group send primtives **ogsend** and **ugsend**. Elapsed time is the length of time during which a sender remains blocked after invoking an **ugsend** or **ogsend** routine. Elapsed time for these primitives depends on a number of factors, including the underlying system's workload, number of secondary managers for the group, and whether the IMMEDIATE_REPLY bit (explained in Section 4.3.1) is set in msgtype. The elapsed time is also dependent on the speed of the processor and the type of network interface. For our measurements we used four 16 MHz 68020 based SUN workstations, each connected to a 10Mbs Ethernet interface with 32 receive buffers.

The measurements were made by performing OGSEND and UGSEND message transmission N times and dividing the total elapsed time by N to obtain a reasonably accurate estimate for a single operation. Table 4.1 gives the elapsed time for the UGSEND and OGSEND message transmissions as a function of the size of the remote group members. In this case the process which invokes the **ogsend** or **ugsend** primitives resides in the host where the primary manager of the group executes.

Table 4.2 is similar to Table 4.1, except that the process which invokes the primitives resides in a host where a secondary manager for that group executes. In both cases the receiving managers acknowledge a message only after the message is delivered to and acknowledged by their local members (i.e.,

Table 4.1

No. of members	ugsend	ogsend
1	8.8	9.5
2	19.2	19.9
3	21.2	21.8
4	23.7	24.0

Elapsed time (milli seconds) for ugsend and ogsend.
Sending process in the same host as the primary manager.

Table 4.2

No. of members	ugsend	ogsend
1	-	-
2	19.2	21.2
3	21.7	22.4
4	23.2	25.1

Elapsed time (milli seconds) for ugsend and ogsend.
Sending process in the same host as a secondary manager.

IMMEDIATE_REPLY bit is off). N is chosen to be 30,000 for both measurements.

The first observation from these figures is that the elapsed time for both primitives doubles when a remote member is added to the group. However, the increase in the elapsed time for additional remote members is not very significant. This behaviour is understandable, because the underlying network is a broadcast network and the time to transmit a group message to one remote site

or multiple remote sites is the same, assuming that the probability of a packet loss is negligible. This may be a valid assumption since the network interface has 32 receive buffers which considerably reduces the chances of losing a packet.

The second observation is that the elapsed time for the UGSEND message transmission is less than that for the OGSEND transmission. However, this difference is not very significant when the process which invokes these primitives resides in the same host as the primary manager. The reason is that the UGSEND message transmission is carried out by either the primary or secondary manager for the group executing locally, but the OGSEND message is sent to the primary manager which may be executing in a host different from that of the sending process.

4.8 Chapter Summary

The proposed mechanism is structured into a set of cooperating processes. Each host runs a group server process. Processes wishing to create or to join the group invoke the appropriate group management stub routines which in turn send appropriate requests to the group server. When a group is created, a primary manager for that group is created in the initial member's host. when a process joins the group from a host where neither the primary nor secondary manager for this group executes, a secondary manager for this group is created in that host. Both the primary as well as the secondary managers maintain group management information necessary to coordinate group management and group communication activities. When a message is sent to a group, the proposed mechanism makes sure that the message is delivered to all the managers of the group each of which will then deliver the message to the local members of the group. In order to improve the concurrency of the manager processes, each of them employ some worker processes such as couriers, probers and vultures. Any

incomplete message transmission as a result of primary or secondary manager failure will be detected and completed by the proposed mechanism. Network partition results in subgroups. Communication within these subgroups will be ordered and atomic. When the partitioned network remerges, the mechanism detects this and merges the subgroups to form a single group. Mesaurements on the elapsed time for ogsend and ugsend indicate that ordered group send has some overhead compared to unordered group send.

Chapter Five
Conclusions

This work describes the design and implementation details of a reliable group communication mechanism. A group communication mechanism is reliable if it has the two aspects of reliability: full delivery and correctness. In order to ensure full delivery the sender must know the identities of the members of the group. If the underlying network supports multicast or broadcast then the message can be first multicast in a datagram fashion to the recipients and then retransmitted on a one-to-one basis to those that failed to receive it the first time around. However, if the underlying network supports only unicast, then the message can be sent on a one-to-one basis to the recipients. Issues related to correctness are atomicity, order and survivability. Atomicity ensures that every message sent to a group will be delivered to all operational members or to none of them. Order guarantees that messages will be delivered in the same sequence to all the operational members. Survivability assures continuous operation despite process, host or network failures.

Full delivery does not necessarily guarantee correctness. Partial delivery may occur if the sender fails in the middle of a transmission. Also, if the group membership is dynamic, it is difficult for the sending process to maintain an up to date membership information. Furthermore, in a system with multiple senders and multiple receivers, a message sent from a sender may arrive at a destination before the arrival of a message from another sender; however this order may be reversed at another destination. This violates the order prpoerty.

In order to provide the reliable properties transparent to the application processes, some form of coordination is necessary in the underlying group communication mechanism. If the underlying mechanism provides a process which

knows the identities of the group members, the messages from multiple senders can be funneled through this process thus ensuring full delivery as well as order. In order to ensure atomicity and survivability in case of the failure of the funnel process, this process may be replicated at different sites. In the proposed mechanism, each group has a primary manager (funnel process) which is replicated at all member sites (secondary managers). If the primary manager fails, a new primary manager is selected from among the secondary managers. The new primary manager will finish any incomplete message transmission initiated by the failed primary manager. This guarantees atomicity and survivability. The ordered messgage transmission is called OGSEND.

Both primary and secondary managers maintain manager member lists which contain the pids of all the managers. The manager member list is updated only by the primary manager. Whenever a secondary manager joins or leaves the group, the primary manager is notified. The primary manager updates it's manager member list and informs the secondary managers to update their lists as well. Each manager (primary as well as secondary) also maintains a local group member list which contains the pids of the members of the group local to their respective hosts. Each manager is responsible for updating it's local group member list. When the primary manager receives a message, it sends the message to all the secondary managers each of which in turn delivers the message to their local members. This requires less space, less network traffic and reduced code complexity than the case of replicating the entire membership information in the primary and all the secondary managers.

Some applications do not require an ordered delivery but only atomic delivery. The unordered message transmission is called UGSEND. These messages need not be funneled through the primary manager. Since all the secondary

managers know the pids of all the managers of the group, a UGSEND message is first sent to a secondary manager for the group residing in the sending process' host which then transmits this message to all other managers. If a secondary manager for the group is not executing in the sending process' host then the message is sent to the primary manager which then transmits it to the rest of the managers. If a secondary manager fails, the primary manager detects this and finishes any incomplete UGSEND message transmission initiated by the failed secondary manager. Also, the primary manager deletes this secondary manager's pid from it's manager member list and informs the other operational secondary managers to update their lists as well.

As mentioned earlier, when the primary manager fails, a new primary manager is selected from among the secondary managers. Secondary managers use a succession list selection scheme to select the new primary manager. In this scheme the oldest secondary manager forces the younger ones into accepting it as the new primary manager. This can be easily done in the proposed mechanism since all the secondary managers have the same manager memberlist ordered by the time they joined the group.

Network partition creates subgroups with the same multicast address. This causes problems when the network remerges. However, the proposed mechanism detects this and merges these subgroups together to form a single group.

Some performance measurements were made on the group send primitives **ogsend** and **ugsend** provided by the proposed group communication mechanism. From the measurements, one observes that the ordered group send takes more time to complete than the unordered group send when the sending process does not reside in the same host as the primary manager. This overhead is due to the fact that the message has to be funneled through the primary manager

residing in a different host, whereas UGSEND messages are sent to the local secondary manager which then transmits them to the rest of the managers of the group.

The mechanism has been implemented on the V Kernel running on four SUN-3/50 workstations interconnected by an Ethernet. The system works as expected and some performance data have been reported in the thesis.

Bibliography

[1]. K.P.Birman and T.A.Joseph, *Reliable Communication in the Presence of Failures*. ACM Transactions on Computer Systems, Volume 5, Number 1, February 1987.

[2]. J.M.Chang and N.F.Maxemchuk, *Reliable Broadcast Protocols*. ACM Transactions on Computer Systems, Volume 3, Number 1, February 1985.

[3] J.M.Chang and N.F.Maxemchuck, *A Broadcast Protocol for Broadcast Networks*. Proceedings of GLOBCOM, December 1983.

[4] S.T.Chanson and K.Ravindran, *A Distributed Kernel Model for Reliable Group Communication*. Proceedings of the IEEE-CS Symposium on Realtime Systems, New Orleans, December 1986.

[5]. D.R.Cheriton, *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier, NY, 1982.

[6]. D.R.Cheriton, *The V Kernel: A Software Base for Distributed Systems*. IEEE Software, Volume 1, Number 2, April 1981.

[7]. D.R.Cheriton and W.Zwaenepol, *Distributed Process Groups in the V Kernel*. ACM Transactions on Computer Systems, Volume 3, Number 2, May 1985.

[8]. F.Cristian, H.Aghili, R.Strong and D.Dolev, *Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement*. Technical Report RJ 4540(48668), IBM, October 1984.

[9]. A.Frank, L.D.Wittie and A.J.Bernstein, *Group Communication on Netcomputers*. Proceedings of the 4th International Conference on Distributed Computing Systems, San Francisco, CA, May 1984.

[10]. H.Garcia-Molina, *Elections in a Distributed Computing System*. IEEE Transaction on Computers, Volume C-31, January 1982.

[11]. H.Garcia-Molina and A.K.Abbot, *Reliable Distributed Database Management*. Technical Report CS-TR-047-86, Department of Computer Science, Princeton

University, August 1986.

[12]. R.Gusella and S.Zatti, *An Election Algorithm for a Distributed Clock Synchronization Program.* Proceedings of the 6th IEEE-CS International Conference on Distributed Computing Systems, Cambridge, MA, May 1986.

[13]. L.Lamport, *Time, Clocks and the Ordering of Events in a Distributed System.* Communications of the ACM, Volume 21, Number 7, July 1978.

[14]. K.Ravindran and S.T.Chanson, *Process Alias Based Structuring Techniques for Distributed Computing Systems.* Proceedings of the 6th IEEE-CS International Conference on Distributed Computing Systems, Cambridge, MA, May 1986.

[15]. F.Schneider, D.Gries and R.Schlicting, *Reliable Broadcast Protocols.* Science of Computer Programming, Volume 3, Number 2, March 1984.

[16]. D.Skeen, *Determining the Last Process to Fail* . ACM Transactions on Computer Systems, Volume 3, Number 1, February 1985.

Appendix A

IPC Primitives of V Kernel

This Chapter describes the IPC operations provided by the V Kernel which is used as the underlying system to build the proposed group communication mechanism described in the thesis. The V Kernel evolved from two previous systems - Thoth and Verex [5]. The V Kernel is referred to as distributed because it's facilities are available uniformly and transparently across multiple machines connected by a local network. Thus, it provides the appearence of a single kernel interface, for the most part successfully masking the existence of multiple machines. A connected set of machines that provides a single V Kernel program environment and name space is called a **V Domain** [6] as depicted in Figure A.1.

The major facilities provided by the V Kernel are processes and communication between processes. In such an environment, services are offered by server processes while client processes communicate with servers using the inter process communication (IPC) primitives to negotiate and receive services. The sender and receiver of an IPC activity are specified by their process identifiers

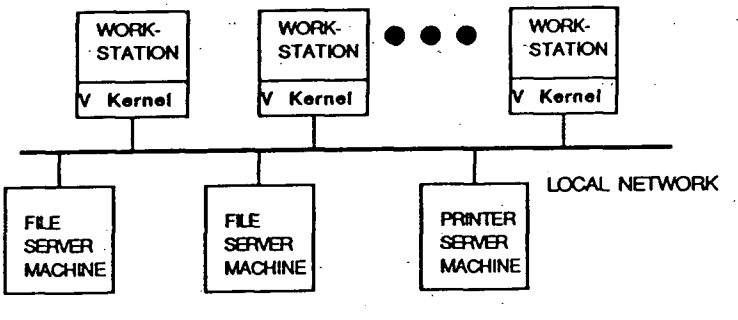


Figure A.1 V domain of local network-connected machines.

(pids). The most common communication scenario is as follows. A client process executes a **Send** operation to transmit a message to a server process and is suspended. The message eventually causes the server's execution of the **Receive** operation to be completed. The server executes a **Reply** to send a reply message to the client. This **Send-Receive-Reply** activity is referred to as a message transaction an example of which is shown in Figure A.2. The following section describes the Send operation in detail. Sections A.2 and A.3 describe the Receive and Reply operations respectively. While explaining the Send, Receive and Reply operations, we assume that the processes involved in the communication activities do not reside on the same host.

A.1 Send Operation

A process wishing to send a message to another process invokes Send(msg, pid), where msg is a pointer to a 32 byte message to be transmitted to a process whose id is specified in pid. The process invoking the Send primitive is suspended and it resumes operation when the receiver replies or if the send operation fails.

When the Send primitive is invoked, the sender is suspended. The sender's kernel transmits a SEND inter-kernel packet with the message embedded to the

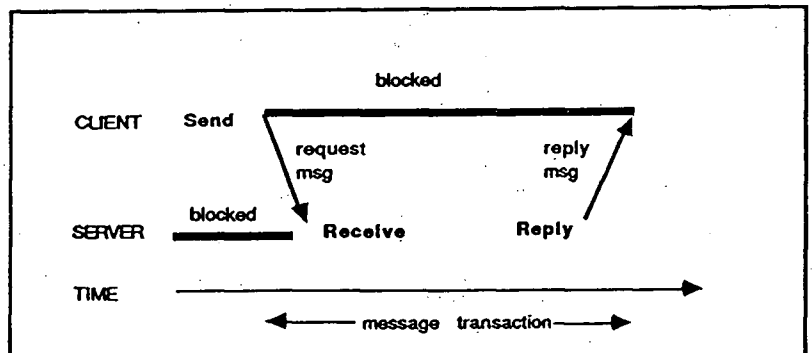


Figure A.2 Send - receive - reply message transaction

receiver's kernel. The underlying hardware assures that all the network interfaces but the one at the desired destination discard the message. When a SEND packet is received by the receiver's kernel, it first checks for the existence of the receiver. If the receiver does not exist, the receiver's kernel replies to the sender's kernel with a NON_EXISTENT_PROCESS message. The sender's kernel then unblocks the sender and informs it of the outcome. However, if the receiver is alive, then the receiver's kernel queues the message first-in-first-out in the receiver's message queue.

If the receiver does not reply to the sender within a specified time period, the sender's kernel retransmits the SEND packet to the receiver's kernel. If the receiver's kernel finds that there is already an identical SEND packet queued up in the receiver's message queue, it discards the incoming packet and replies with a BREATH_OF_LIFE message to the sender's kernel which resets it's retransmission count. If after a number of retransmissions the sender's kernel does not receive any response from the receiver's kernel, it assumes that the receiver's site has failed or the network has partitioned. The sender's kernel therefore unblocks the sender and informs it that the Send operation has failed. Unfortunately the sender's kernel cannot distinguish between site failure and network partition unless this information is available in the underlying communication medium such as in some ring-type networks. Figure A.3 depicts the Send operation in various scenarios.

By invoking the Send primitive, one can be assured that as long as the receiver is alive and the network is not partitioned, the message will be delivered to it's destination. Also it allows the sender to exploit positive acknowledgement and retransmission for reliable delivery or determination of process, host or network failure.

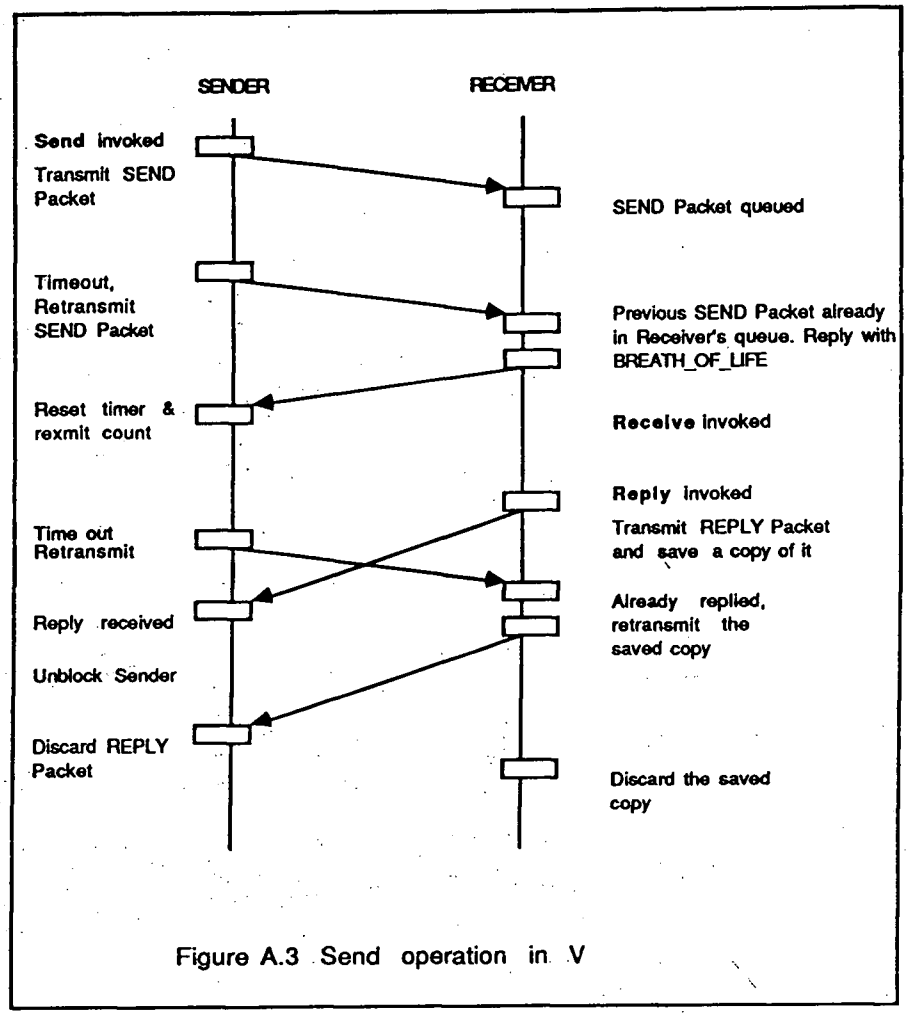


Figure A.3 Send operation in V

A.2 Receive Operation

The V Kernel provides two different primitives for receiving messages; **Receive** and **ReceiveSpecific.** When a process invokes Receive(msg) to obtain a 32 byte message at the location pointed at by msg, it may receive a message from any process in the domain. The receiver's kernel simply suspends the receiver until a message arrives from some sender. On the other hand, ReceiveSpecific(msg, pid) is used to obtain a 32 byte message from a process whose id is specified by pid. When a process (receiver) invokes ReceiveSpecific to receive a message from a particular process (sender), the receiver's kernel suspends the invoker. It first checks if there is already a SEND packet queued

up in the receiver's message queue from the specified sender. If there is none, the receiver's kernel sends a RECEIVE inter-kernel packet to the sender's kernel. When the RECEIVE packet is received by the sender's kernel, it checks whether the sender exists or not. If the sender does not exist, it replies with a NON_EXISTENT_PROCESS message to the receiver's kernel which in turn unblocks the receiver and informs it of the outcome. On the other hand if the sender exists, it's kernel replies with a BREATH_OF_LIFE message. On receiving this message, the receiver's kernel resets it's retransmission count. If the receiver's kernel does not receive a SEND packet from the sender within a specified time period, it repeats the ReceiveSpecific procedure again. In case the sender's site fails or if the network partitions, the BREATH_OF_LIFE message from the sender's kernel will not be received by the receiver's kernel. Therefore, after a maximum number of retransmissions the receiver's kernel unblocks the receiver and reports that the ReceiveSpecific operation has failed. As in the Send operation, the receiver's kernel cannot distinguish between network partition and host failures. Figure A.4 illustrates the ReceiveSpecific operation.

By invoking the ReceiveSpecific primitive, a process can detect the failure of another process or the site on which the process resides. Thus, using the ReceiveSpecific primitive, the **vulture** scheme described in Section 3.3.2 can be easily implemented.

A.3 Reply Operation

A process may only reply to a process from which it has received a SEND packet. This is necessary to maintain tight synchronization of the **Send-Receive-Reply** activity. When a process (replier) invokes Reply(msg, pid) to reply with a 32 byte message pointed at by msg to a process specified by pid, it's kernel first checks to see whether the replier had already received a SEND

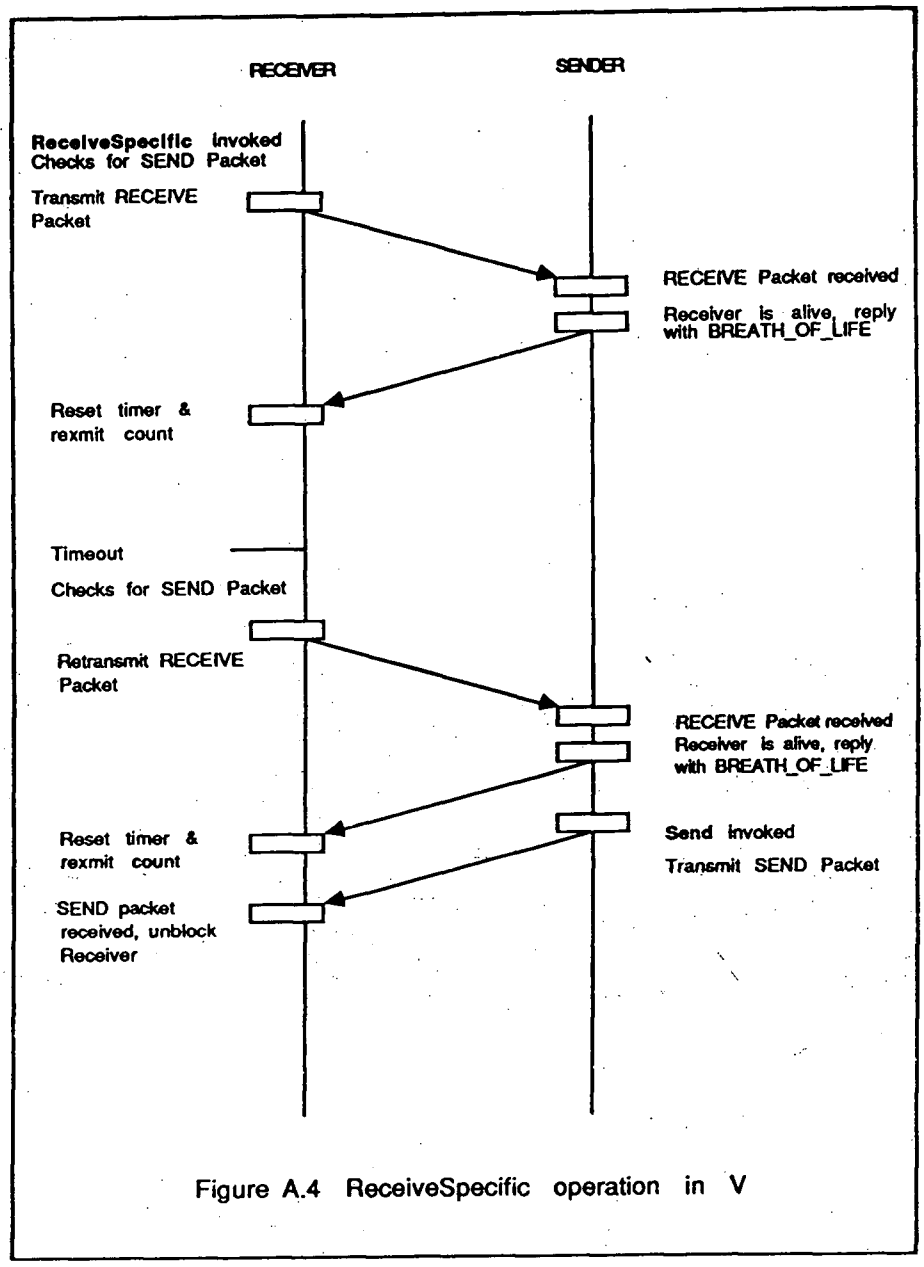


Figure A.4 ReceiveSpecific operation in V

packet from this process. If the replier did not receive a SEND packet from the specified process, then the Reply operation fails. Otherwise, a REPLY inter-kernel packet is sent to the specified process' kernel. We have already seen, under the Send operation, how the sender's kernel keeps transmitting the SEND packet until a REPLY packet is received or timeout, whichever occurs first. After the REPLY packet has been sent, a copy of it is kept in the replier's kernel for

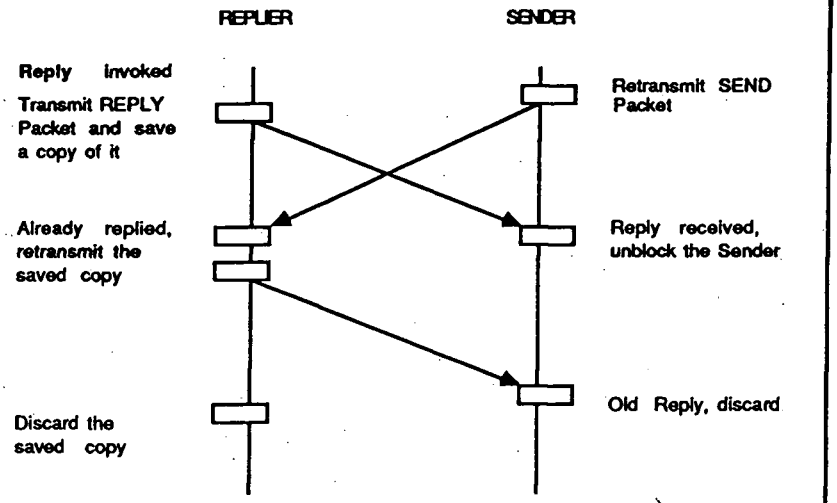


Figure A.5 Reply operation in V

sometime. The replier's kernel retransmits this saved copy in response to retransmitted SEND packets. The copy is discarded after a specified time period. The Reply operation is illustrated in Figure A.5