Reliable Multicast Communication For Parallel Computation in Distributed Environments

by

Frances F. Liu

B.Sc., The Second Branch of Beijing University, Beijing, China, 1983

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Gholamali C. Shoja, Supervisor (Department of Computer Science)

[redacted]

Dr. Michael R. Levy, Departmental Member (Department of Computer Science)

[redacted]

Dr. Nikitas J. Dimopoulos, Outside Member (Department of Electical & Computer Eng.)

[redacted]

Dr. Peter F. Driessen, External Examiner (Department of Electical & Computer Eng.)

Supervisor: Dr. Gholamali C. Shoja

Abstract

Multicasting is a useful and efficient communication method for distributed and parallel applications. This thesis is concerned with the design and implementation of reliable multicast interprocess communication (IPC) mechanism for supporting distributed computations in an environment where certain types of failure could occur.

The need for reliable multicast communication mechanism arises in applications that are distributed to achieve parallel processing, resource sharing, data availability and reliability. Many of these applications require different levels of reliability when using multicast communication. Although most local area networks support broadcast communication, such broadcast facilities are rather low-level, unreliable and cannot be fully utilized by application programs.

This thesis presents a high level, decentralized protocol for achieving reliable multicast communication in distributed environments. The protocol guarantees that messages are received by all the operational receivers or by none of them(full delivery and atomicity). It also ensures that the messages sent from the senders will be delivered in the same order to all the receivers(global ordering).

The protocol introduces the simple but powerful concept of *current status vectors* which is attached to every outgoing message, together with a logical timestamp and a sequence number. These mechanisms enables each node to ensure atomicity and global ordering independently and in a distributed manner.

The proposed reliable multicast communication protocol has been implemented in REM(Remote Execution Manager). Performance data for benchmarks using reliable multicasting is compared with those obtained under unicast communication.

Examiners:

Dr. Gholamali C. Shoja, Supervisor (Department of Computer Science)

Dr. Michael R. Levy, Departmental Member (Department of Computer Science)

Dr. Nikitas J. Dimopoulos, Outside Member (Department of Electical & Computer Eng.)

Dr. Peter F. Driessen, External Examiner (Department of Electical & Computer Eng.)

Contents

List of Figures

List of Tables

Acknowledgements

This thesis could not have been completed without the guidance of my supervisor, Dr. Gholamali C. Shoja. I thank him for his patience in teaching me, and for his continuous support and encouragement during the past two years.

I would like to take this opportunity to thank my fellow students in the department of Computer Science. Special thanks to Robert Side, for his patience in explaining the REM system to me; to Eric Davies, for proof reading my first draft of the thesis; and to many other students who have in one way or other provided all the assistance and moral support needed in making this thesis possible.

And above all, I like to thank my mother, my brother and his family in China for their constant love, support and encouragement.

Last but not least I thank the University of Victoria Graduate School and Computer Science Department for providing the financial support.

Chapter 1

Introduction

A distributed system is generally viewed as multiple computational elements (nodes) interconnected by a communication network. One of the design goals of distributed systems is to provide more reliability and availability than centralized ones. To achieve this goal it is necessary to replicate computations and databases at different nodes so that computations can continue despite node failures. In a non-distributed setting, a failure affects only the the user of the crashed program or machine. In a distributed system, however, the effect of a crash can ripple through a large number of machines.

In order to access or update data on different machines, and to maintain the consistency among copies of the data , a set of distributed cooperating processes residing on different nodes, need to communicate and synchronize with each other. Therefore, efficient inter-process communication and synchronization mechanisms form the essential parts of any distributed system.

1.1 Inter-process communication in distributed systems

1.1.1 The problem

Inter-process communication in a distributed system is more complex than in a system which contains shared memory. The consistency control and synchronization among nodes of a distributed system must be achieved by message passing. The hardware in a distributed environment allows a processor to send messages to other processors. The operating system provides the facility that allows a process on one machine to send message to a process on another machine. However, these facilities are rather low-level abstractions for inter-process communication.

A common high-level abstraction for inter-process communication is the *remote procedure call* (RPC), introduced by Birrell and Nelson [Birrell 84]. A process communicates with another using an interface that looks just like a call to a local procedure. The advantage of this abstraction is that it simplifies distributed programming by making communication with a remote process look like communication within a process. Its disadvantage, however, is that it is limited to two-way communication, namely communication between a calling process and a called process. Therefore, remote procedure calls are most useful in distributed programs that fit the "client-server"model. Also, RPC is not the most convenient abstraction when a distributed program is composed of a number of processes that have a high degree of inter-dependence. An example of such a program would be a server that, for reasons of fault-tolerance or load sharing, is implemented as a group of processes on a number of sites. Each member in the group must ensure that its actions are consistent with what the other members are doing, so they need to communicate with one another.

In this kind of environment, not only a one-to-one (unicast) process communication facility is needed, but a facility that enables a process to send a message to a set of processes would be desired. That is, a broadcast or a multicast facility.

The mechanisms for inter-process communication can be classified by the characteristic of the sender and the receiver(s) of messages, i.e. unicast, broadcast and multicast.

1.1.2 The mechanisms

1.1.2.1 Unicast communication

Communications are traditionally discussed in terms of a single transmitting entity (the *sender*) and a single receiving entity (the *receiver*). The sender transmits a message to a receiver, which thus receives a copy of the message. This type of communication is known as **unicast** communication. Unicast communication facility is normally provided in an operating system and some protocols may be included to ensure a certain degree of reliability.

1.1.2.2 Broadcast communication

Broadcast is the delivery of a message to all the destinations in the entire system. It is widely supported by most local area networks. It is efficient since a message directed onto the network may be received by any machine connected to the network but has few uses at the application level [Mockapetris 83], as essentially no information is relevant to all machines in a network. In many cases, application programs are only communicating with processes residing on a subset of the machines.

1.1.2.3 Multicast communication

Multicast is the delivery of a message to some specified subset of the possible destinations. Multicast is a natural extension of point-to-point communication in which a user can communicate with a set of selected users simultaneously. The primary motivation for multicast communication stems from the wide range of applications that require it, both in a local area network(LAN) and in a broadband packet network. Some important applications include fault-tolerance, distributed processing, distributed databases, multiuser games, teleconferencing, etc. In some fault tolerant applications it is necessary to store multiple copies of the same file on separate file servers. To keep those multiple copies of a file updated and consistent the same message(data) has to be sent frequently to every machine where a copy is kept.

Unfortunately, the support for multicast communication is not found in many distributed systems; at best, it is often emulated using unicast protocols. Consequently, considerable network bandwidth and local processing time is consumed for large messages.

1.1.3 Reliable multicast communication

The requirement for reliable multicast communication arises in applications that are distributed to achieve fault tolerance, parallel processing, resource sharing, data availability and reliability. A common example is updating replicated copies of a distributed database maintained on different hosts. In some applications, it is required that messages must not only be delivered to all the destinations but must also be delivered in the same order. Although broadcast/multicast facilities can be found in many LANs, they are not totally reliable. Since most application programs can not improve the reliability at the network level, it is the user's responsibility to ensure

the reliability of the communication at the application level.

We consider a multicast communication mechanism to be reliable only if it satisfies the following properties: full delivery, atomicity and global ordering.

1. *Full delivery*:
   Every non-faulty member in the multicast group receives the message provided the sender does not fail in the middle of the transmission.

2. *Atomicity*:
   If one member does not get the message, the other members will not process the message or send another multicast message until the particular receiver finally receives the message.

3. *Global ordering*:
   Messages are delivered to all the members in the same order.

Detailed description of these features are given in Chapter 3

1.2 System models

1.2.1 Distributed system

We consider a system to be distributed if it consists of a collection of processes residing on a number of hosts and communicating by messages. More than one process may reside on the same host. Figure 1.1 shows our model of a distributed system.

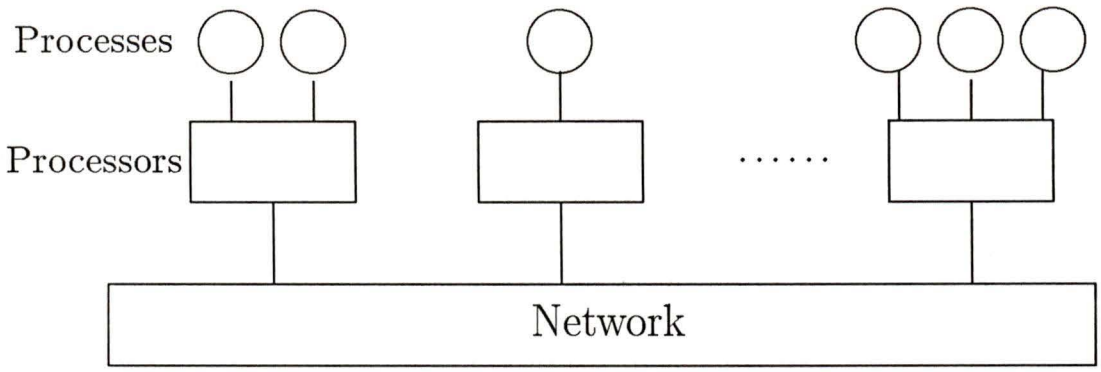


Figure 1.1: **Distributed system model**

1.2.2 Broadcast Network

The underlying network which interconnects a distributed system may or may not support broadcast/multicast. In most Wide Area Networks(WANs) and some non-broadcast Local Area Networks(LANs), broadcasting can be simulated by sending multiple unicast messages. Wall's thesis [Wall 80] presents several mechanisms for performing efficient broadcast and multicast delivery in point-to-point networks.

We assume the underlying network to be a broadcast LAN. Broadcast networks have become a popular topology as a result of their simplicity, high performance and very low cost. Broadcast networks use a single physical medium to carry a message to all recipients. Ethernet [Metcal 76] was chosen in our implementation. Ethernet gives the impression that it provides reliable delivery at the hardware level since the error rate is very low. The actual loss of packets in the Ethernet is as low as one in two millions. But still, it is possible that messages may be lost, duplicated or delivered in the wrong order due to insufficient buffer space, contention in the network or an undetected collision.

The central component of a broadcast network is the shared communication chan-

nel. Typically, this is a passive transmission medium such as a coaxial cable, fiber-optic cable, or simply space (in the case of radio broadcast network). Processors are connected to the channel through links. Links contain active components such as signal transceivers, network interface units, and buffer memory for storing incoming and outgoing messages. Figure 1.2 shows a typical distributed configuration based on a broadcast network.

Our reliable multicast communication protocol also applies to the systems built on networks that do not support broadcast. In such an environment, logical broadcast or multicast can be achieved by routing algorithms at the kernel level. When a kernel performs all the message routing, the complexity of broadcast/multicast is hidden from the user.

1.3 Objectives of the thesis

In this thesis, we present a method for design and implementation of a reliable multicast communication protocol for parallel computing in a distributed environment. The protocol guarantees that all the members in a multicast group will receive the messages. The send operation is atomic and the order of messages is the same on all recipients.

The objectives of the thesis are:

- to describe the need for a reliable multicast facility in a distributed environment;
- to present our proposal and method for achieving reliable multicasting;
- to describe our development of a general set of high-level multicast communication primitives;

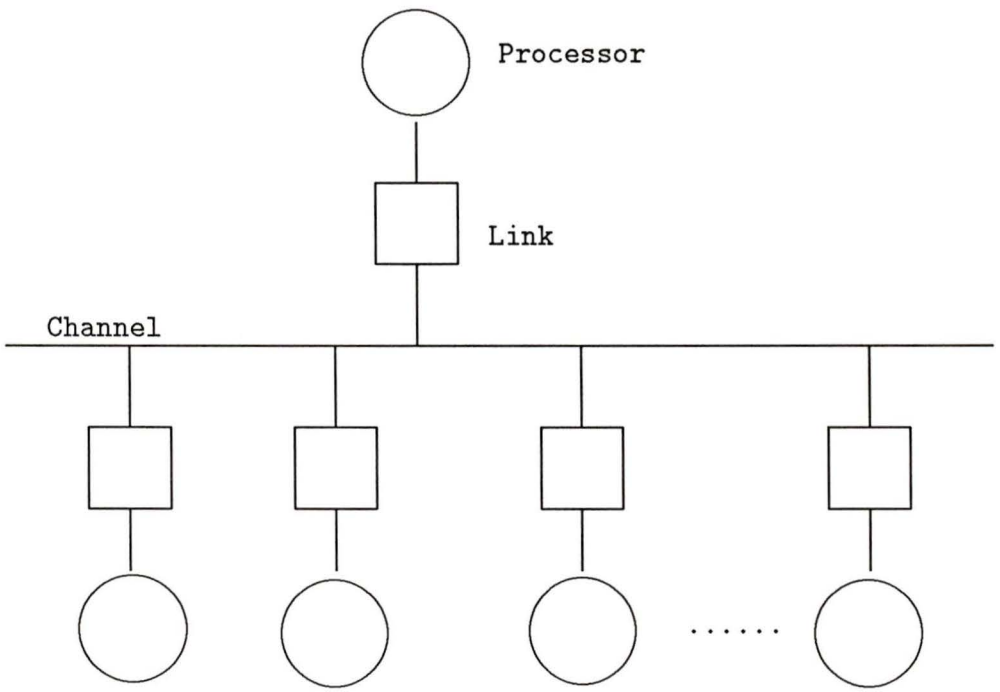


Figure 1.2: **Broadcast network**

- to describe our implementation of a multicast communication facility developed for the REM's distributed computing environments;
- to present the test results when using our developed multicast facility for parallel computation with REM.

1.4 Organization of the thesis

The remainder of the thesis is organized as follows. Chapter 2 contains a review of recent research contributions in the area of reliable multicast protocols. Chapter 3 presents our reliable multicast protocol and shows how the three reliability properties are ensured. Chapter 4 describes the design and development of a set of general purpose reliable multicast communication primitives and layered structure of implementing the protocol in a distributed environment. Chapter 5 describes the design and implementation details of how the protocol was integrated into the Remote Execution Manager (REM) environment. Chapter 6 presents the results of benchmarks to evaluate the performance of the reliable multicast communication in REM. Chapter 7 concludes the thesis by summarizing our accomplishments and discussing directions for future research.

Chapter 2

Background

The problem of reliable broadcast/multicast protocols has been dealt with by many researchers in the literature in various contexts ([Cristian 86], [Chang 84], [Kaashoek 89], [Navaratnam 88] and [Birman 87]). We review some of the previous work in this area which are relevant to our research.

2.1 Message diffusion approach

Cristian et al. [Cristian 86] proposed a protocol for the reliable and ordered delivery of a message to all hosts in a distributed system (i.e. broadcasting). It assumes a point-to-point network, bounded network transmission delay and a very strong clock assumption, i.e. the clocks of correct processors measure the passage of time accurately, and are synchronized.

The protocol is also based on a simple information diffusion technique, in which:

1. A correct sender sends a message on all its outgoing links.

2. When a host receives a new message, it forwards that message on all the outgoing links.

A set of protocols are introduced to tolerate different classes of faults. We only discuss one of them which is based, as in our protocol, on the assumption of fail-stop processors [Skeen 83].

- Ordering property: The messages are timestamped with physical time to enable ordered delivery and to detect duplicates. This is made possible in a distributed environment by the clock assumption. In every correct processor, messages are delivered in the order of their generation times (or timestamps). If two messages are generated at the same clock time, they are delivered in increasing order of their senders identifier. All messages delivered from all senders are delivered in the same order at all correct receiving nodes.

- Atomicity property: To ensure that any message broadcast at clock time t by some processor s and received by at least one correct processor r is received by every correct processor q, before q delivers any message broadcast at clock time t, its delivery is delayed for a period of time Δ determined by the intersite message delivery latency.

- Cost and efficiency

  The performance of Cristian's protocol is dependent on the accuracy with which the clocks are synchronized and the operating system's task scheduling mechanism. The protocol is efficient in terms of message traffic but is costly in terms of memory space needed on each node as during the *information diffusion* phase, every node has to keep N (number of nodes in the network) copies of the message.

2.2 Centralized (primary-secondary) approach

Some researchers have tackled the atomicity and ordering problems using a *centralized* approach [Navaratnam 88], [Kaashoek 89], [Chang 84]. Instead of broadcasting/multicasting a message to all the receivers, all the messages are first sent to one of the receivers which then, acting as a *controller* or *primary manager*, broadcasts/multicasts the message to the rest of the receivers in order. Therefore, the atomicity and global ordering can be easily guaranteed.

2.2.1 Examples

Chang et al. [Chang 84] proposed a protocol which, like Cristian's work, is responsible for the delivery of a message to all the hosts in a distributed system. However, their philosophy is also a *centralized* one where the messages are funnelled through a coordinator called the 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 for reliable delivery on the receiver hosts. 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 it sends the token host a negative acknowledgement for the missing message. The role of token host is rotated among the operational hosts to provide reliability and resiliency.

In Kaashoek and Tanenbaum's approach [Kaashoek 89], whenever a sender wants to send a broadcast message, it sends the message to its local kernel which then sends to the *sequencer* through a point-to-point protocol. The *sequencer* broadcasts the messages it receives to the network in a FIFO manner; global ordering is guaranteed at all nodes. Figure 2.1 shows the system configuration for such an approach.

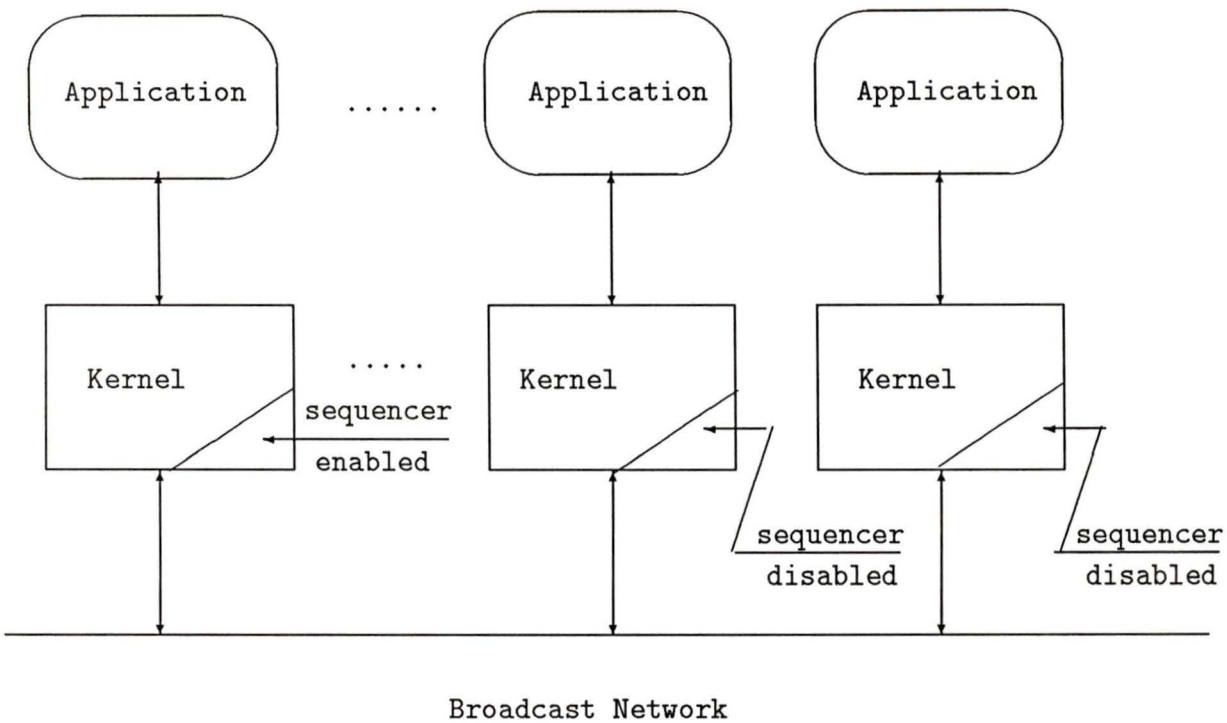


Figure 2.1: **System structure of a centralized approach**

2.2.2 Discussion

The *centralized* approach is simple to understand and is an easy method for achieving atomicity and global ordering, but it also has the following disadvantages:

- The controller may become a bottleneck in the system when there are many broadcast/multicast requests.
- Large buffer space is needed on the host which acts as a controller in order to queue up the incoming messages before sending them out and keeping them for retransmissions.
- A failure detection and recovery mechanism is required to ensure reliability and survivability when the controller fails. The system can either let the pre-assigned *secondary(standby)* controller take over the job, or use a selection algorithm to elect a new controller. Moreover, the "history" on the primary controller has to be copied to the new controller.

2.3 Building the multicast facility in operating system kernel

In some distributed systems, even though the underlying network supports broadcast/multicast, the application programs cannot access the facilities merely because the operating system does not provide the functions. Therefore, some distributed operating systems are designed to support broadcast/multicast by integrating one-to-many interprocess communication in the system kernel, i.e. building a kernel interface to the broadcast capability of a local area network and thus permitting efficient utilization of this capability. The V-System [Cheriton 84], [Cheriton 85a] and

T-Shoshin system [Vuong 89] are examples of this approach and provide group communication and management primitives for the users. Processes may join groups in which a member may multicast a single message to the entire group. Their model however only provides a kernel-level unreliable "best efforts" delivery for multicast messages. Hence, it is left up to the upper communication and/or application layers to incorporate the degree and kind of desired reliability.

Multicast communication can be integrated into UNIX by introducing an additional *socket type* [Ahamad 85]. The implementation involves some minor changes to UNIX system. Since it is based on an unreliable datagram service and offers no reliable delivery or reply mechanism, users must build a protocol at the application level if they require the reliability property.

Although the implementation of this approach is relatively easy and provides a neat and transparent interface to users, not many applications are allowed to touch the system kernel or even the network driver(like in T-Shoshin). Therefore, protocols which can be implemented at the application level are needed.

2.4 Building an interface on top of the operating system

In this approach, a multicast facility is provided to application programs without modifying the underlying network or operating system kernel.

2.4.1 ISIS system

ISIS [Birman 87] is a distributed programming toolkit developed at Cornell University. It introduced a set of protocols for reliable broadcast[1] communication, each providing a different degree of reliability. Users can choose a protocol based on the particular application requirements. For instance, *ABCAST* for atomic broadcast, *CBCAST* for *causal* broadcast, etc.

The basic ISIS multicast protocol is designed around a point-to-point model. Each process in a group maintains a two-way reliable data stream with every other process in the group.

A two-phase protocol is used to ensure atomicity and ordering properties. 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 member receives a message, 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 the order of increasing priority. This guarantees order. However, the sender has to reliably communicate with all the members twice before the message is delivered. Also, it is implemented at the cost of memory space since each recipient of a message must store the message until it is notified that the message has been delivered to all the message's destinations.

[1]ISIS' *broadcast* is actually multicast by our definition.

2.5 Summary

In this chapter we have reviewed several previous works addressing the problems similar to the ones that we wish to tackle, i.e. reliable multicast communication in distributed environment. Some of the proposed protocols require a significant number of control message exchanges; e.g., token transfer messages in [Chang 84] and priority-number messages in [Birman 87] under normal (no failure) conditions. Some protocols do not require control messages but assume that clocks are synchronized and working processors are always connected (e.g., [Cristian 86]). While in the centralized approaches such as [Navaratnam 88], [Kaashoek 89], and [Chang 84], atomicity and ordering properties are based on explicit *sequencers*. Additional election and recovery mechanisms are needed for fault-tolerance.

Chapter 3

Proposed Method and Protocol

3.1 Multicast group management

In multicast communication, the concept of process groups is fundamental. A set of distributed cooperating processes is referred to as a process group. A process may communicate with a group of processes as a single logical entity using the group identifier GID. For a static group where the group membership never changes, maintaining the group membership is trivial and can be built into the underlying system. However, when the multicast group has a dynamic size, the reliable delivery of message to all members of the group needs some coordination mechanism.

Some reliable multicast/group communication mechanisms are built into the distributed operating systems where process group management is provided in the system kernel. Examples can be found in V system [Cheriton 85a], [Cheriton 85b], T-shoshin system [Vuong 89], and also in [Navaratnam 88]. In our design, we do not make any assumptions about the underlying operating system. In fact, the software is built on top of SUN OS4.0.3 which is a derivative of BSD UNIX system. As a

result, we had to design our own process group management mechanisms.

3.1.1 Assumptions

We make the following additional assumptions:

- Clock assumption:
  Processors do not share memory or maintain closely synchronized clocks. Processes only have access to local clocks.

- Failure assumptions:

  1. Fail stop: We assume that when processor machines fail, they simply cease execution without taking any incorrect or malicious action [Skeen 83].
  2. The communication system can also fail: it can lose and duplicate messages, or deliver them out of order.

3.1.2 Definitions

Our group communication model uses the following terminology:

- **Group**: An association among a set of processes for cooperation to solve a problem.

- **Group communication**: A location-transparent way to communicate with the members of a group.

- **Group identifier**: In multicast group communication, groups are addressed by group identifiers(GIDs). GID is unique and there are no two groups sharing the same GID.

- **Group membership**: Each member in a multicast group is assigned a member identifier. Each member has a list of members in the group and their member identifiers.
- **Group size**: total number of members in a group. For a dynamic multicast group, the group size can vary.
- **Group view**: A snapshot of the membership of a group at some (logical) instant in time.

3.1.3 Characteristics

In our design, a multicast group has the following characteristics.

- Any process can create a multicast group.
- Any member in a multicast group can both send and receive multicast messages.
- The size of a group is dynamic.
- Members of a group will receive all the messages sent to the group.
- A process can use *join* or *leave* primitives to join or leave an existing group.
- Only the creator of a group can remove the group.

3.1.4 Properties

Our multicast group communication satisfies the following properties.

- A message sent to a multicast group is delivered to all the operational members of the group or none of them. This is to satisfy the atomicity of our protocol.

- If two messages α and β are sent to a multicast group by the same sender and α is initiated before β, then α arrives before β at all the members in the group.
- If two messages α and β are sent to a multicast group by two different senders then the messages are received by all the members in the group in the same order, i.e. either α, β or β, α.

3.2 Reliable multicast communication protocol

In our protocol, multicast communications are carried on in *multicast groups*. A multicast group is a set of processes residing on different hosts. A process can create, join or leave a multicast group and can send or receive multicast messages in the group. Only the creator of the group can delete the group. The group is destroyed when the creator fails or terminates. A process must join a group before sending to or receiving from that group. A group manager process is needed to maintain this kind of dynamic group membership. We discuss the implementation issues in Chapter 4.

3.2.1 Properties of reliable multicast communication

In Chapter 1, we gave the definition of a reliable multicast communication protocol and its three properties, i.e. full delivery, atomicity and global ordering. In this section, we discuss those properties in detail.

3.2.1.1 Full delivery

Full delivery ensures that a message sent to a multicast group will be delivered to all the non-faulty members in the group provided the sender does not fail in the

middle of the transmission. If the underlying network is a broadcast network, the sender broadcasts (or retransmits if necessary) the message to all the members in a datagram fashion. If the underlying network supports only unicast, the sender has to send the message to each member in the group individually. Therefore, as long as the underlying network supports reliable one-to-one IPC facility, a multicast communication mechanism can ensure full delivery in multicast groups.

3.2.1.2 Atomicity

Atomicity ensures that every message sent to a multicast group is either delivered to all the members of the group or to none of them. If one member does not get the message, the other members will not process the message or send another multicast message until the particular receiver finally receives the message or is assumed dead by the rest of the members.

It is necessary to distinguish the difference between full delivery and atomicity. Full delivery ensures the message is sent to all members as long as the sender remains functional 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 a partial delivery. Partial delivery is harmful in many applications. The atomicity property guarantees that if a message is delivered to one operational member of a group then the message will be delivered to the rest of the operational members as well.

In our protocol, atomicity is achieved by using an **acknowledgement diffusion** approach. It is based on the consideration that the underlying network only supports unreliable broadcast transmission:

1. The sender sends a message to a multicast group.

2. On reception of a message, each recipient broadcasts its acknowledgement to the entire group.

3. If a recipient has collected all the acknowledgements from the other recipients, it knows the message has been received by every member in the group so the message can be delivered to the application program.

4. If the sender fails to collect all the acknowledgements from all the group members, it times-out and retransmits the message.

5. When a member receives a retransmission, it broadcast its acknowledgement to the whole group.

6. A *second round confirmation* message is sent to the group as soon as the sender collects all the acknowledgements.

3.2.1.3 Global ordering

The global ordering property ensures that messages are delivered in the same order to all the members.

Atomicity is not enough since there are no guarantees on the order in which different multicasts will be delivered. 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 message 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. When more than one member can send multicast messages simultaneously, the global ordering of messages at receivers is hard to achieve.

Most published reliable broadcast/multicast protocols which provide the global ordering property adopt the *centralized* approach [Kaashoek 89] [Navaratnam 88]

[Chang 84]. Everyone must send their messages to a particular node and then all the messages are broadcast/multicast to the rest of nodes in the order of message arrival at that particular node, hence, global ordering is guaranteed. This approach needs an extra mechanism of recovery to handle the problem when the particular node fails.

An obvious approach for determining the order is to *timestamp* the events. If global clock synchronization is provided, physical timestamps can tell us the ordering. In a distributed system where processes coordinate their actions by sending messages to one another and do not have a global clock for synchronization, events can only be partially ordered in terms of the **happened before** relation which was introduced by Lamport [Lampor 78]. Each process maintains an integer value, initially zero, which is periodically incremented, e.g. once after every atomic event. This value is added to every outgoing message as its timestamp. An acyclic event order, denoted $\rightarrow$ reflects the dependence of events.

We assume that sending and receiving a message are events in a process and adopt Lamport's notation $\rightarrow$ to denote the **happened before** relation, i.e., event p happened before event q is denoted as $p \rightarrow q$. Figure 3.1 is an example of Lamport's algorithm with logical timestamps.

In multicast communication, messages are exchanged among many processes, not just between two processes. Lamport's single integer value timestamp is insufficient for determining the message order. Instead, we use a vector to represent the logical time when an *event* happens in a process. The use of the vector timestamp is addressed in 3.2.2.1 and section 3.2.9.

The event $msend_p(m)$ denotes the multicast transmission of the message m by process p; the receive event is denoted by $mrecv_p(m)$. We distinguish the event of *receiving* a message from the event of *delivery*. Event $mrecv(m)$ is the arrival of a message at a host while $deliver(m)$ is to pass the message to the application process

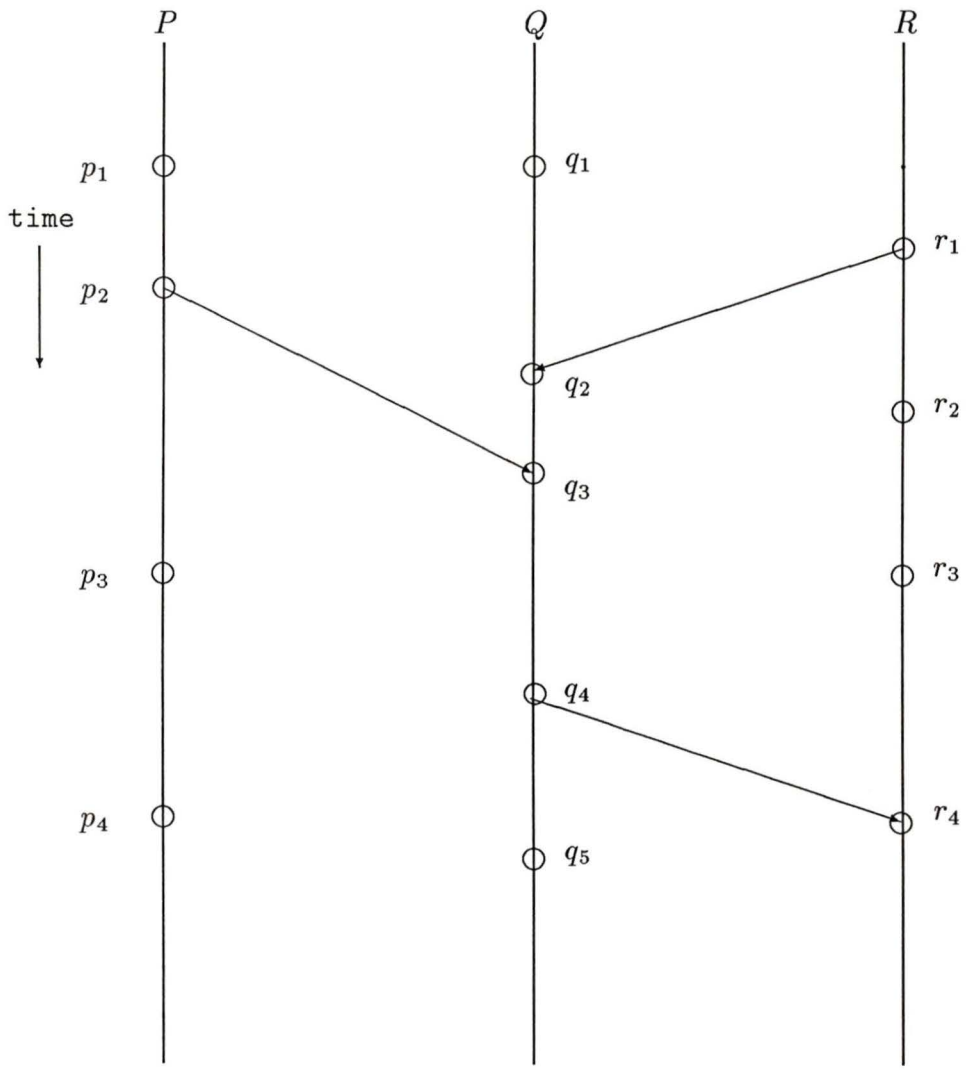


Figure 3.1: **Lamport's single integer logical timestamps**

when certain conditions are satisfied.

For a message m,

$$msend(m) \rightarrow mrecv(m)$$
$$mrecv(m) \rightarrow deliver(m)$$

When process p_i sends a message m and process p_j sends a message m' to the same multicast group, assuming $i < j$, if $msend(m) \rightarrow msend(m')$ then $deliver(m) \rightarrow deliver(m')$ at all members. If the two events $msend(m)$ and $msend(m')$ are concurrent, then every member follows $deliver(m) \rightarrow deliver(m')$.

3.2.2 Data structure

Before giving the detail of the protocol, we present the data structure used in the reliable multicast software.

A multicast process group is a set of processes $P = \{p_1, p_2, ..., p_n\}$ with disjoint memory spaces. n is the number of processes in the group. Initially, this set is static and known when the group is created. Later the membership of the group can be changed and the size of the group varies when members join or leave dynamically.

Each member in a multicast group has a unique member identifier (member_id) which is an integer. The member_ids are predefined when the group is created. The creator of the group has the lowest number. When a new process joins an existing group, it will get a member id which is group size + 1. The group size (GSIZE) is increased by 1.

3.2.2.1 Logical timestamps

We use a vector $TS_{p_i}[1..n]$ to represent a logical time. p_i is a process in a multicast group, i is the local process member identifier assigned to the process. The vector length n is the group size which is changeable. $TS_{p_i}[k]$ is the logical clock value on process p_k. When a message is sent out from a process, a TS vector is attached to the message as its logical timestamp.

We adopt Fidge's partially-ordered clock [Fidge 88] to construct a logical timestamp for each send/receive event. Fidge's clock is described in Appendix A. We attach a logical timestamp on every outgoing message (including acknowledgements and retransmissions).

The TS vectors are maintained by the following rules:

R_1 : When p_i starts execution, TS_{p_i} vector is initialized to zero.

R_2 : For each primitive event($msend(m)$ or $mrecv(m)$) in a process p_i, $TS_{p_i}[i]$ is incremented by 1.

R_3 : The current value of the entire logical timestamp vector is sent to the receiver(s) with every outgoing message.

R_4 : When process p_j receives a message m from process p_i containing timestamp TS_{p_i}, p_j modifies its local vector time in the following manner: p_j sets the value of each entry in its local timestamp vector to the maximum of the two corresponding values in the local vector and in the remote vector received. The element corresponding to the sender is a special case; it is set to one greater than the value received, but only if the local value is not greater than that received. i.e.,

$\forall\, k \in 1..n$:

$$TS_{p_j}[k] = max(TS_{p_j}[k], TS_{p_i}[k]) \text{ when } k \neq i$$
$$TS_{p_j}[k] = max(TS_{p_j}[k], TS_{p_i}[k] + 1) \text{ when } k = i$$

Vector timestamps are compared as follows:

$$TS_{p_i} \leq TS_{p_j} \textbf{ iff } \forall k : TS_{p_i}[k] \leq TS_{p_j}[k]$$
$$TS_{p_i} < TS_{p_j} \textbf{ if } TS_{p_i} \leq TS_{p_j} \text{ and } \exists k\text{: } TS_{p_i}[k] < TS_{p_j}[k]$$

It can be shown that given message m(from p_i) and m'(from p_j),

$$msend(m) \rightarrow msend(m') \textbf{ iff } TS_{p_i} < TS_{p_j}.$$

Detailed proof can be found in [Fidge 88].

Figure 3.2 is an example of multicast communication with logical timestamps.

3.2.2.2 Multicast current status

To ensure full delivery and atomicity, a *multicast current status* for a process p_i, denoted CS_{p_i}, is used to keep track of the state of a member during multicast communications. $CS_{p_i}[1..n]$ is a boolean vector of length n, p_i indicates the member id of a local process and n is the group size which is changeable. CS_{p_i}[k] is a boolean value, '1(true)' or '0(false)'. 'k' ($1 \leq k \leq n$) corresponds to the numbering of the processes in the group.

Initially, the CS vector is set to zero which indicates that all members in a multicast group are not sending or expecting any messages. A *stable state* of process p_i is represented as

$$CS_{p_i}[k] = 1, 1 \leq k \leq n.$$

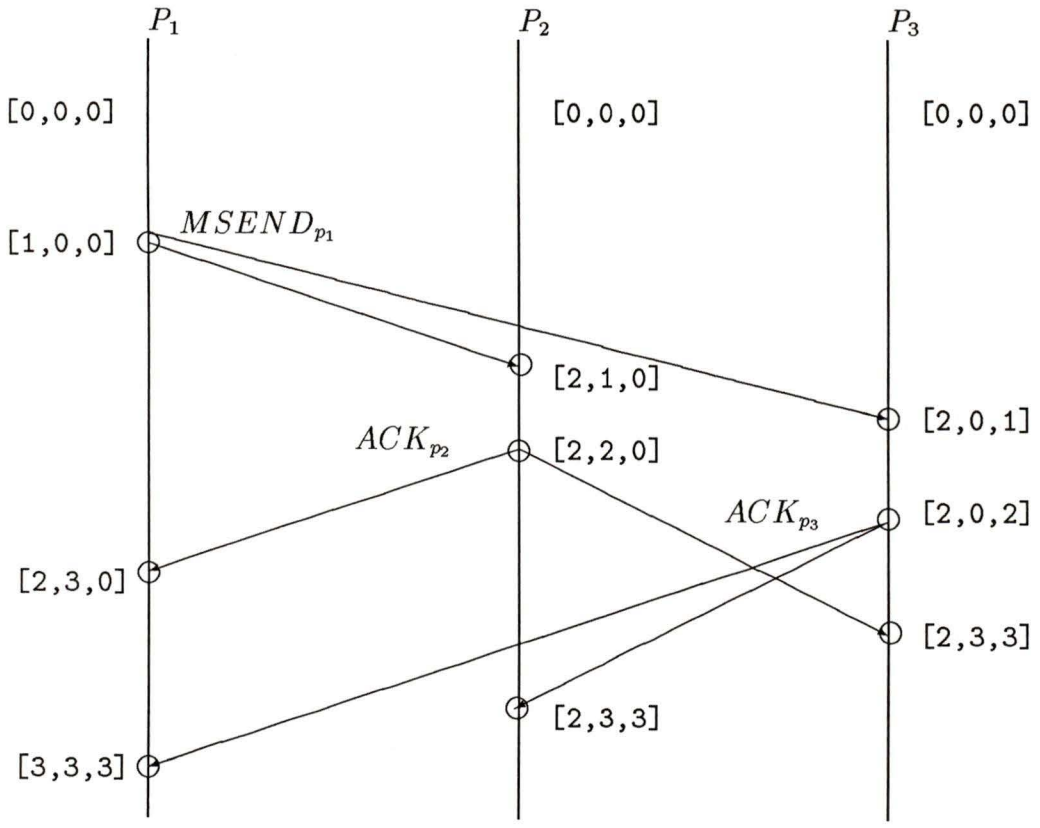


Figure 3.2: **Logical timestamp vectors in multicast communication**

When a member sends out a message (data or acknowledgements), its current status changes to *true*. The corresponding element in CS vector becomes '1'. This updated vector is sent to the rest of members with the message. When a member receives a message, it updates its own current status vector based on the remote CS vector. If $CS_{local}[i]$ is '0' but $CS_{remote}[i]$ is '1', then $CS_{local}[i]$ should be changed to '1' except when i is equal to the member id of the multicast message source. Only when a group member gets a multicast message or retransmission of the same message from the sender, can the corresponding element in its CS vector be changed to 1. Otherwise, if the certain element in a remote CS vector is 1, but the local value is 0, indicating it has not received the original multicast message, the value should remain 0.

The CS vectors are maintained by the following rules:

R_1 : When p_i starts execution, CS_{p_i} vector is initialized to zero.

R_2 : For each primitive event $msend(m)$ in a process p_i, $CS_{p_i}[i]$ is set to 1(true).

R_3 : The current value of the entire current status vector is delivered to the receiver(s) with every outgoing message.

R_4 : When process p_j receives a message m from process p_i containing CS_{p_i}, p_j modifies its *multicast current status* vector in the following manner:

- When m is the original multicast or retransmission message:
  $\forall\, k \in 1..n$:

  $$CS_{p_j}[k] = CS_{p_j}[k] \vee CS_{p_i}[k]$$

- When m is an acknowledgement: $\forall\, k \in 1...n\ k \neq i$:

  $$CS_{p_j}[k] = CS_{p_j}[k] \vee CS_{p_i}[k]$$

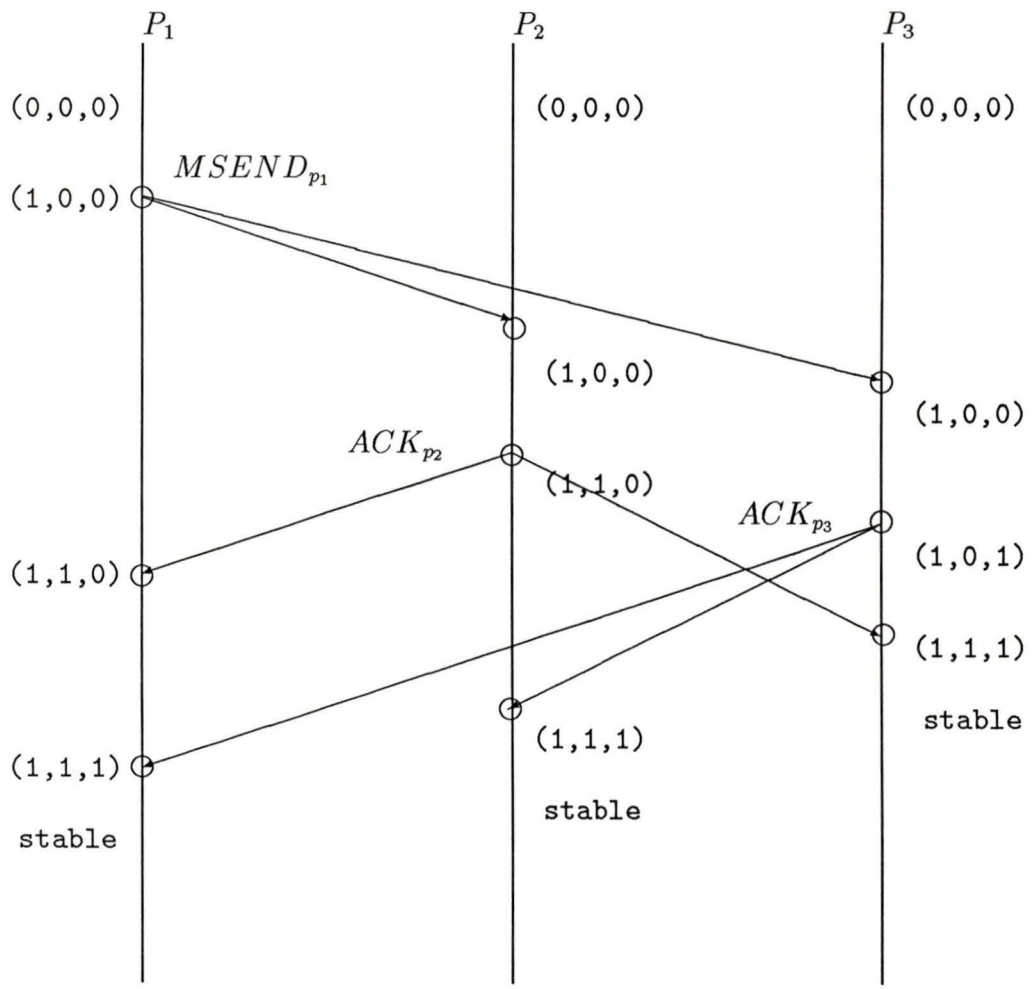


Figure 3.3: **Current status in a multicast communication (error-free)**

Figure 3.3 shows how CS vectors work in an error-free multicast communication. To differentiate from TS vectors, we use round brackets to represent CS vectors.

3.2.2.3 Sequence numbers

In our decentralized protocol, any member in a multicast group can send multicast messages to the group. To keep track of incoming messages from different sources, on a process p_i, we use a vector $SEQ_{p_i}[1..n]$ to record the sequence numbers of the incoming multicast messages. $SEQ_{p_i}[k]$ indicates the sequence number of the message sent by process p_k. $SEQ_{p_i}[i]$ is the number of outgoing multicast messages sent from process p_i. Initially, this vector is set to zero.

3.2.3 Sending out a multicast data message

When the multicasting module gets a *msend* request from a process, it puts the message into the outgoing message queue first and then checks whether it is in a stable state before sending out the message. By *stable* we mean the sending process has collected all the acknowledgements for the previous multicast message, or if the process was a receiver last time, it can be a sender only when it is sure that no other member in the group is waiting for acknowledgements. This is done by checking the current status vector (CS). When all the elements in the vector are true, the member is in a stable state. The member then initializes the CS vector to zero and broadcasts the message to the group with a local sequence number, a new CS vector and a new logical timestamp (TS vector). The sending process will set a timer and enter a blocking state to wait for acknowledgements. If it fails to collect all the acknowledgements within a certain period of time, retransmission is initiated.

3.2.4 Retransmission scheme

To ensure delivery, each time a multicast message is sent out, a timer is set for that message. Retransmissions are necessary if a message remains unacknowledged by at least one receiver for a period of time(estimated round-trip delay + processing time).

Figure 3.4 shows how CS vectors work when an acknowledgement is lost resulting in retransmission of the multicast message.

When a receiver receives a duplicate retransmitted message (same sequence number as the previous message), the receiver must send a multicast acknowledgement to the whole group even though it has received the message. This is to let the other hosts have a better global knowledge of what's going on in the group. If the duplicated message was caused by a network error, it will be discarded.

3.2.5 Receiving a multicast data message

After sending a multicast message to a group of processes, the sender will be blocked waiting for the acknowledgements before it can send the next message. Upon receiving a multicast message, the receivers echo the arrival of the message by sending acknowledgements. Here we have several decisions to make as how and where to send the acknowledgements.

3.2.6 Acknowledgement schemes

3.2.6.1 Separate and piggybacked acknowledgements

An acknowledgement message can be sent in a separate packet or piggybacked in outgoing messages. Piggybacking is efficient and can reduce the number of the messages

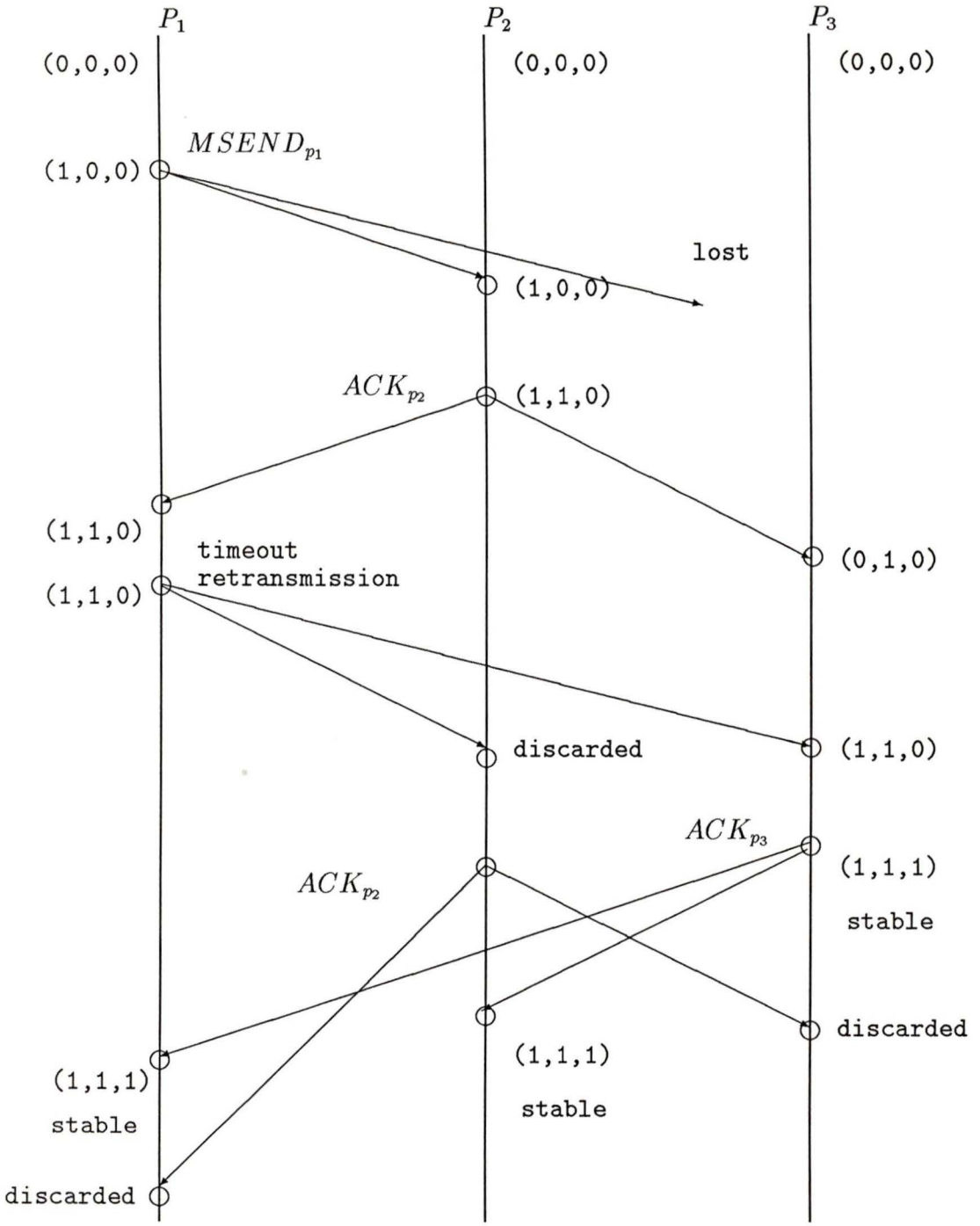


Figure 3.4: **Current status in a multicast communication (with retransmission)**

exchanged in the protocol but it is based on the assumption that there is always a subsequent message. This assumption is not always true in a real environment where the frequency of sending messages is rather random in different nodes. As a result, a system deadlock may occur when a node does not send out messages for a long time. To prevent this from happening, each node sends a dummy message after receiving a certain number of messages without sending any [Kaashoek 89].

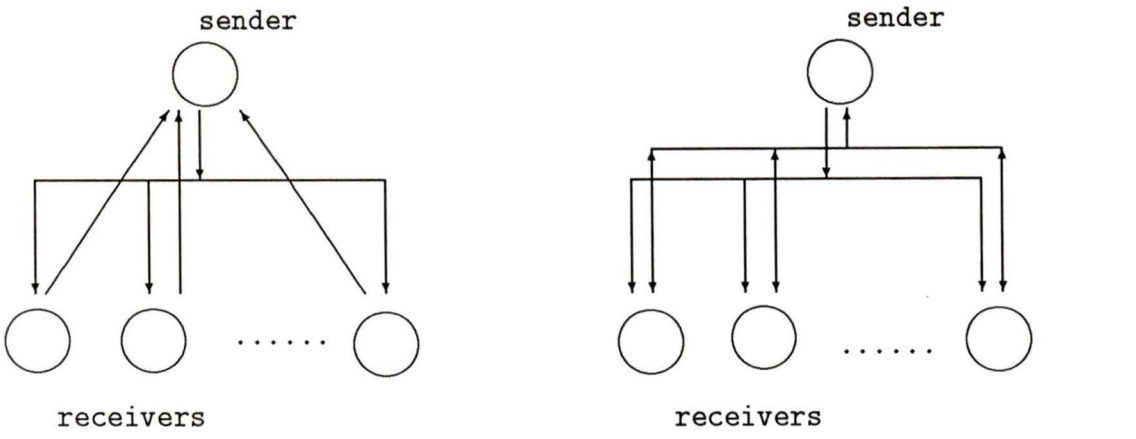
We chose to use separate acknowledgements because they are more suitable for multicasting environments. Compared to the multicast messages (which can be large block of data or a file), acknowledgement messages are very small – just about the size of a message header, and transmission overhead is not very significant.

3.2.6.2 Positive and negative acknowledgements

Positive acknowledgement is a popular technique to achieve reliability of communication. The receiver sends an acknowledgement whenever it receives a message. It is the sender's responsibility to ensure that the receiver indeed receives the message. If it fails to get an acknowledgement from a receiver after a certain period of time, it assumes that the message was lost and a retransmission is invoked.

In contrast to this synchronous approach, the negative acknowledgement (NACK) scheme does not block the sending process, that is, the sending process does not wait for feedback from a receiver. The receiver is entrusted with the responsibility of monitoring the received messages. It requests a retransmission from the sender once it finds that a message is out of sequence.

We use positive acknowledgement to ensure that every receiver gets the multicast message. Each of the receivers echo a confirming message to the sender (and to the rest of receivers) as soon as they receive the message. We adopt this method based



(a). one-to-one acknowledgements (b). multicast acknowledgements

Figure 3.5: **Two schemes of sending acknowledgements**

on the consideration of atomic communication where all the members in a multicast group are tightly inter-dependent. A sender cannot go ahead and send the next message before it is certain that all the receivers have received the current message.

3.2.6.3 One-to-one and broadcast acknowledgement

Acknowledgements can be sent in one of the following methods:

1. Each receiver sends an acknowledgement to the sender on a one-to-one basis.

2. Each receiver broadcast its acknowledgement to the entire group instead of to the sender specifically.

   Figure 3.5 shows the two schemes.

   Many multicast protocols, [LeBlanc 85] [Crowcroft 88] [Wong 85], the acknowledgements of a multicast message are sent as unicasts by each of the receivers to the sender. This is based on the assumption that the underlying network may

not be a broadcast network, in which case multicasting the acknowledgements to the entire group puts a heavy load on the network.

In our protocol, it is appropriate to use the second scheme for the following reasons:

- We assume the underlying network is a broadcast network. When sending a message, it is put on the broadcast bus regardless of being a multicast message or a unicast message.
- Besides the sender, a receiver is also interested in whether the rest of receivers have received the multicast message. After receiving a message and collecting all the corresponding acknowledgements sent by other receivers, a receiver can go ahead and process the message.

Just as the multicast messages can get lost, the acknowledgements multicast by a receiver can be lost too. The sender may fail to collect all the acknowledgements and a receiver may also face the problem that one or more acknowledgements from other members are lost during transmission. We discuss these two cases separately.

Figure 3.6 shows how CS vectors work when one of the acknowledgement is lost at the sender. The acknowledgement sent from P_3 to P_1 (sender) is lost. The sender will timeout and retransmit even though all the receivers received the message. The receivers will then discard the duplicated message (detected by sequence number), and send out a multicast acknowledgement.

Figure 3.7 shows how CS vectors work when one of the acknowledgements is lost at one of the receivers. The acknowledgement from P_3 is lost at P_2. This causes P_2 to remain in an unstable state even though the multicast message has been successfully sent in the group. P_2 will be blocked until it gets another

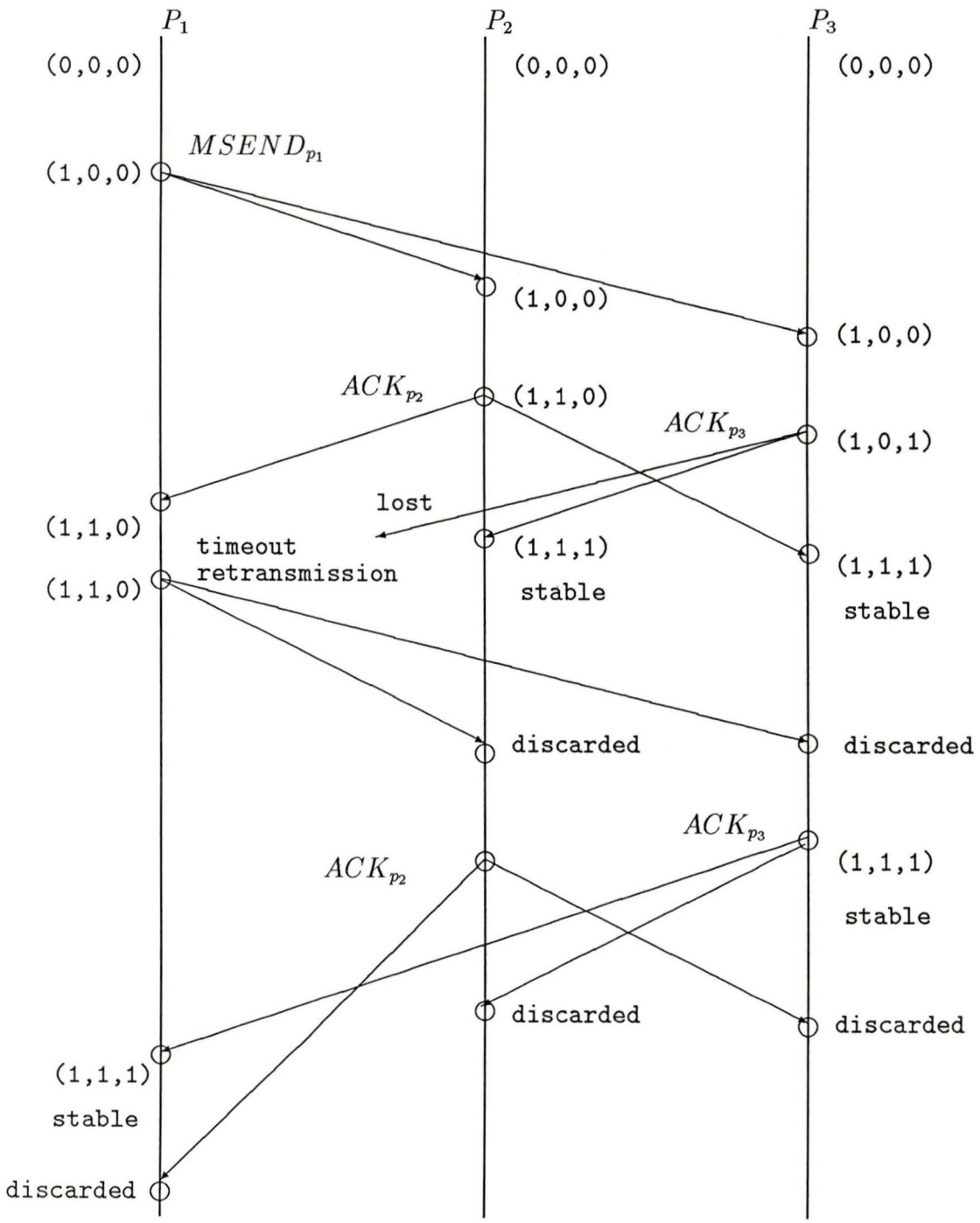


Figure 3.6: **Current status when an ACK is lost at the sender**

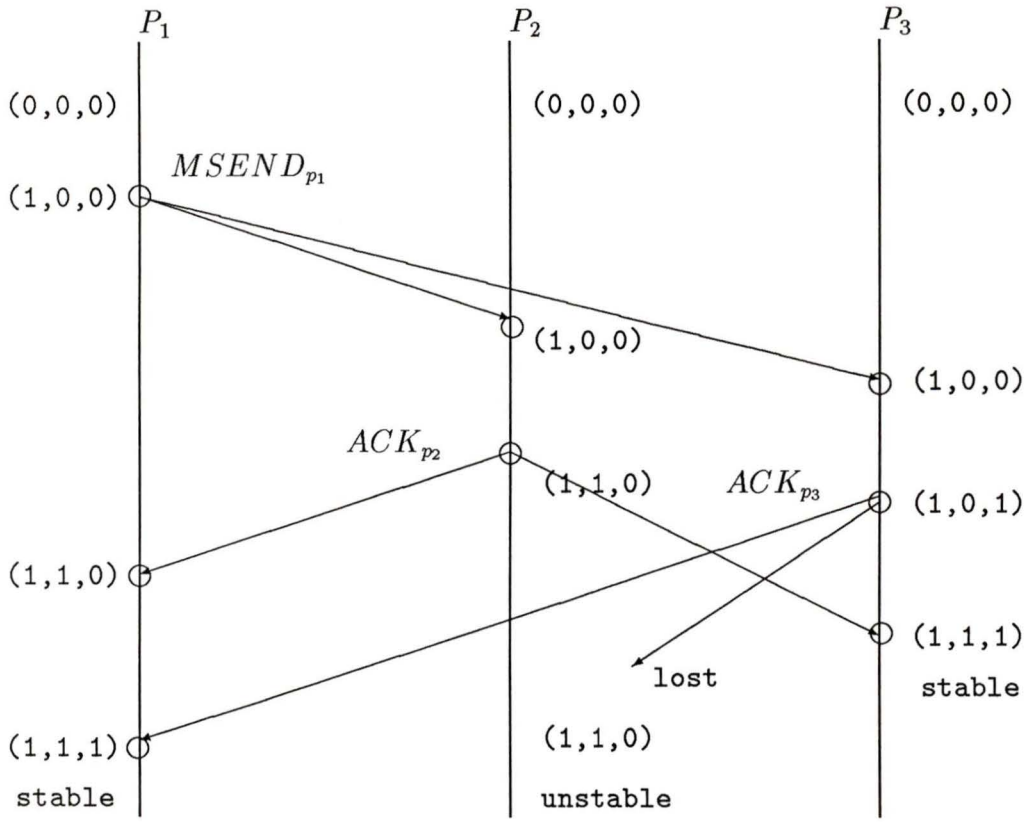


Figure 3.7: **Current status when an ACK is lost at the receiver**

MCAST message from any of the members in the group which means the last *msend* was successful. This will cause the system to be temporarily in an inconsistent state. To solve this problem, we introduce a method called *second round confirmation* which will be discussed in Section 3.2.7.

3.2.7 Second round confirmation

We introduce a method to achieve better atomicity in the presence of acknowledgement losses. Also, it can be used to detect faulty nodes and maintain consistency among the remaining correct members.

3.2.7.1 Unblocking members which are in unstable status

During a multicast communication, once the sender in the multicast group reaches a *stable* state, it multicasts a confirmation message to the entire group. After this *second round confirmation*, every operational member will eventually reach the stable state. See Figure 3.8.

With this second confirmation, we achieve better atomicity and reduce the number of retransmissions with the cost of one more multicast message being sent for every multicast communication.

3.2.7.2 Deletion of the failed members

If a receiver fails or the node it resides on crashes during a multicast communication, the receiving process is terminated based on our failure assumption. All the remaining functional processes in the multicast group will stay in unstable status because they are expecting the response from the failed process. Initially, the sender would timeout and retransmit the message to the group. All the functional members in the group broadcast their acknowledgement with their local CS vectors. The sender now knows that a certain member did not respond to the multicast message and the retransmission. This member will be assumed dead. The sender then sends a confirmation message with a special CS vector where the element corresponding with the failed process is set to "-1" to inform rest of the members in the group that the member is failed. See Figure 3.9

The failed member will be removed from every functional member's membership list.

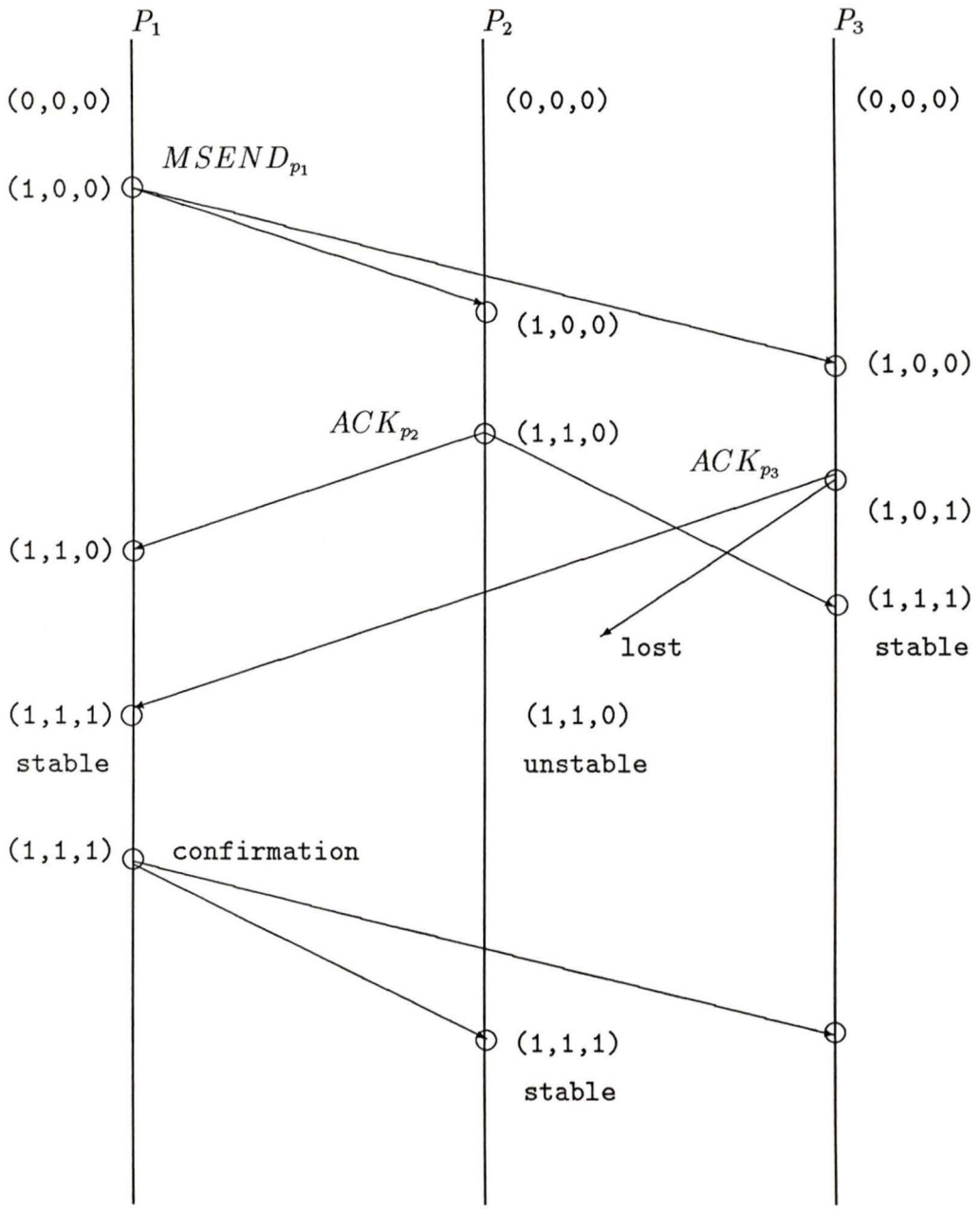


Figure 3.8: **Second round confirmation when an ACK is lost at a receiver**

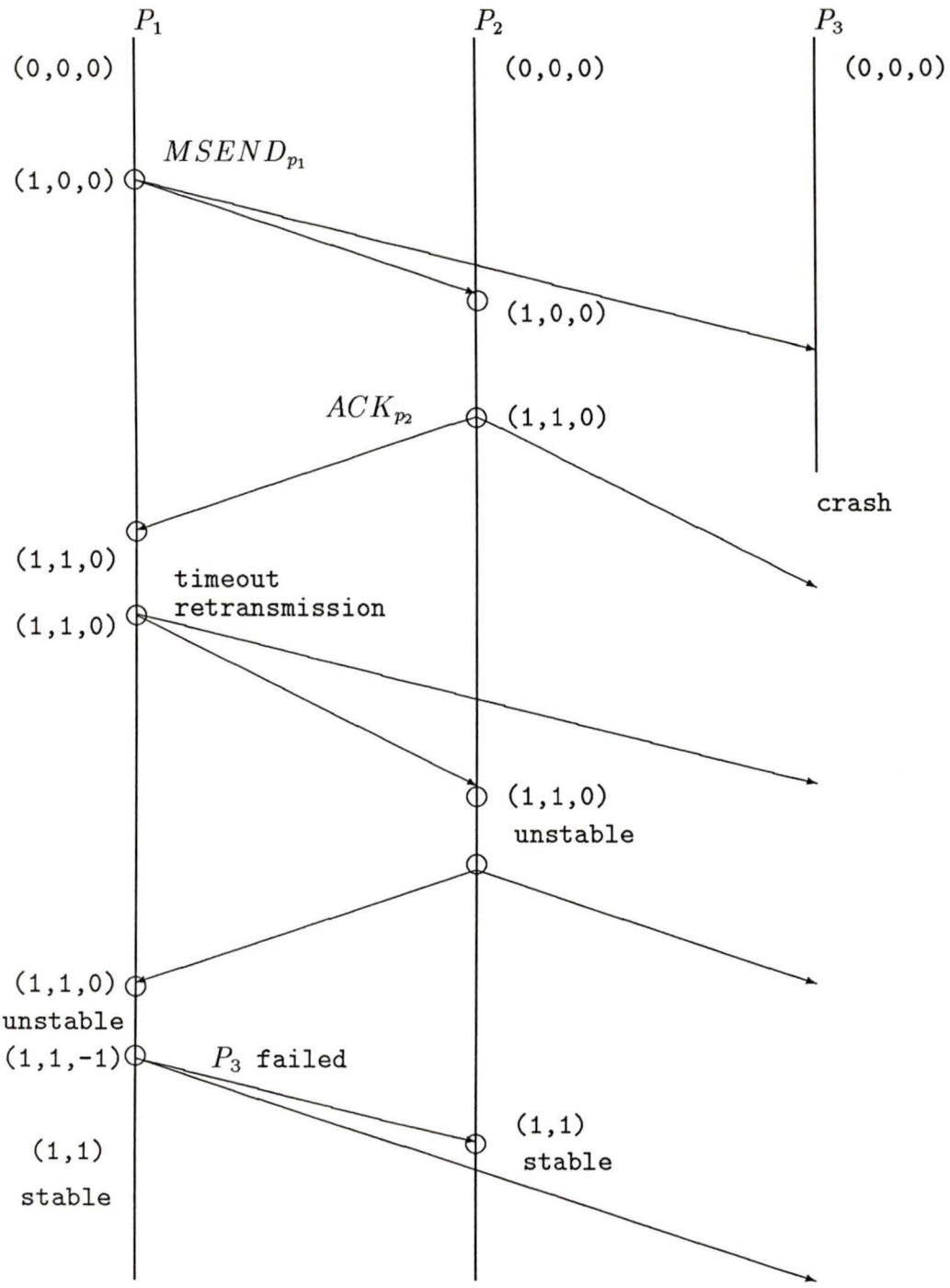


Figure 3.9: **Second round confirmation when a process is failed**

3.2.8 Atomicity guarantee

Atomicity is guaranteed in our protocol by the use of the CS (current status) vector. Every outgoing message is sent with the local CS vector. By getting all the remote CS vectors, a member has a clear picture of what's going on in the system, i.e., who received the multicast message and who missed it. Until collecting all the information from the rest of members, a member stays in an unstable state which forbids it to deliver the message to the upper layer or send its own multicast message. Therefore, the whole system is kept in a consistent state.

When a member fails because the node it resides on crashes, the protocol ensures that the remaining correct nodes get the message and the multicast communication continues.

3.2.9 Global ordering guarantee

We use the logical timestamps to determine the precedence ordering of multicast events. In our protocol, if more than one multicast message is received at a receiver, we will check whether there exists a particular order among them, i.e. if we can get a *happened before* relationship from the logical timestamps attached with the messages.

Consider the action of a process p_j that receives two messages m_1 and m_2 such that

$TS(m_1) \rightarrow TS(m_2)$ which indicates that

$msend(m_1) \rightarrow msend(m_2)$ then

$deliver(m_1) \rightarrow deliver(m_2)$

If $msend(m_1)$ and $msend(m_2)$ are concurrent, i.e. neither

$$TS(m_1) \leq TS(m_2), \text{ nor}$$
$$TS(m_2) \leq TS(m_1)$$

which means neither

$$msend(m_1) \rightarrow msend(m_2), \text{ nor}$$
$$msend(m_2) \rightarrow msend(m_1)$$

We simply sort them according to the increasing sequence of the senders' member identifiers. Assume message m_1 was sent by process p_m and message m_2 was sent by process p_n where $m < n$, then

$$deliver(m_1) \rightarrow deliver(m_2).$$

3.3 Summary

In this chapter, we introduced a reliable multicast communication protocol which satisfies three reliability properties: full delivery, atomicity and global ordering. In the proposed mechanism, full delivery is ensured by positive acknowledgement and retransmission schemes. Atomicity is guaranteed by blocking all the members while checking the *current status* until the entire multicast group reaches a stable status. To achieve global ordering, we adopt Fidge's logical clock and integrate it into our protocol by using *vector timestamps.*

Our multicast message transmission is based on a unreliable datagram fashion of transport mechanism. The protocol tolerants message transmission failures such as missed or duplicated messages.

Chapter 4

Design Issues of the Protocol

This chapter describes the design philosophy of the proposed reliable multicast communication mechanism.

Although the error rate of the underlying network is very low, it is still possible that a packet may be corrupted due to noise on the cable. Also, a packet may be lost because of a receiver being too slow, or due to insufficient buffer space on the receiver host. Reliable transport protocols like TCP/IP guarantee that the unicast messages are delivered from sender to the receiver correctly. This property cannot be used in broadcast/multicast communication. Nevertheless, the multicast communication software is built entirely on top of the system kernel, i.e., at the application level, since we are not allowed to touch the system kernel or build the multicast communication protocol at data link layer.

The multicast communication software is designed in a modular fashion in order to be clear and portable. Figure 4.1 shows the structure of the multicast communication software.

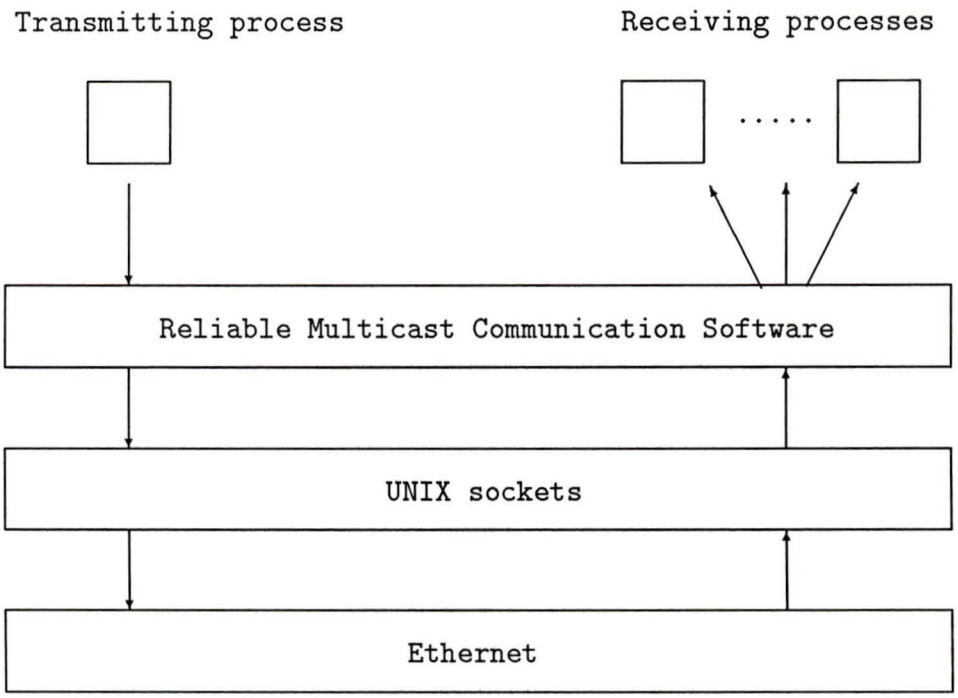


Figure 4.1: **Software structure for a reliable multicast communication**

4.1 The protocol and group management layer

In this layer, a set of primitives for reliable multicast communication and multicast group operation is provided to the user. Users can send to and receive from the reliable multicast communication interface. The reliable message transmission procedure is transparent to users. These primitives are intended, for most part, to be unrelated to any specific network or distributed system. Chapter 5 shows how these primitives can be implemented in REM, a remote execution manager.

4.1.1 The primitives

We propose six primitives for the multicast interface. The primitives are divided into two broad categories: (a) multicast communication primitives, and (b) multicast group management primitives.

4.1.1.1 Multicast communication primitives

(a) *MSEND* primitive: The *send* primitive allows a process to send a message to a multicast group. The message delivery is guaranteed by the built-in protocol which is transparent to the sending process.

ReturnCode = *msend*(group_id, message_pointer, message_len);

group_id: is a multicast group identifier.

message_pointer: pointer to the record structure of a message.

message_len: length of the message.

ReturnCode: an indication as to the success or failure of the primitive.

(b) *MRECV* primitive: Multicast reception.

ReturnCode = *mrecv*(group_id, message_buf, buf_size);

group_id: is a multicast group identifier.

message_buf: buffer to put the received message in.

buf_size: size of the buffer.

ReturnCode: an indication as to the success or failure of the primitive.

4.1.1.2 Multicast group management primitives

(a) *MCREATE* primitive: Before sending a multicast message a process must be member of a multicast group, either by having joined an existing multicast group or by creating a new one. We assume each group has its own group identifier which is unique.

ReturnCode = *mcreate*(group_id, group_size);

group_id: a multicast multicast group identifier;

group_size: number of member processes. The creator is included.

ReturnCode: an indication as to the success or failure of the primitive.

(b) *MJOIN* primitive: A process can join an existing multicast group to become a member by using the *mjoin* primitive and giving a specific group id. The process is added in the process_list of the group.

ReturnCode = *mjoin*(group_id);

group_id: is a multicast group identifier.

ReturnCode: a value indicating the success or failure of the *join* primitive. The process is added in the process_list of the group.

(c) *MLEAVE* primitive: A process can leave a multicast group by using the *mleave* primitive and giving the id of the group. A 'farewell' message will be sent to the entire group. Each of the remaining members will remove the leaving member from their membership list. Once a process leaves a group, it will not be able to send or receive any message to or from the group.

ReturnCode = *mleave*(group_id);

group_id: is a multicast group identifier.

ReturnCode: a value indicating the success or failure of the *mleave* primitive. The process is deleted from the process_list of the group.

(d) *MREMOVE* primitive: The creator of a multicast group can remove the group by using *remove* primitive and giving the id of the group.

ReturnCode = *mremove*(group_id);

group_id: is a multicast group identifier.

ReturnCode: a value indicating the success or failure of the *mremove* primitive.

4.1.2 Timing the outgoing messages

A timer is set once a multicast message is sent out. If the timer expires before all acknowledgements are received, the multicast massage is re-broadcast to the node(s).

The timeout interval is an important parameter in the protocol. If the timeout interval is too long, the protocol performance may suffer due to retransmission of lost messages being delayed. If the timeout interval is too short, frequent retransmissions may occur.

There are two ways to determine a good timeout interval:

- Fixed timeout:

  It is based on the estimated round-trip delay in the network under the load factors of machines at the creation time of the group..

- Dynamic timeout:

  Dynamically estimating the round-trip time(the interval between the sending of a message and the receipt of its acknowledgement) can improve user throughput and network efficiency. Estimated times are used to determine when retransmission will occur. When the network is under a heavy traffic, the timeout is set longer. This can reduce the number of retransmissions.

Currently we only use a fixed timeout.

4.2 Socket layer implementation

4.2.1 UNIX process communication facility

Berkeley UNIX 4.2 and its derivatives(SUN OS, etc.) provide alternative ways for processes to communicate with each other. User processes may choose inter-machine communication media(socket, RPC, etc.), network protocols(TCP/IP, UDP/IP, etc.), addressing families(UNIX domain or INET domain), and style of communication(stream, datagram, etc.) [Leffler 85]. In particular, user processes may use stream or datagram communication with TCP/IP or UDP/IP protocol. When building a distributed application a user can either use one of the system-supplied interfaces or implement a specialized set of network functions that support the application. The easier choice by far is using a system

supplied interface.

The main protocols available for communication across processors in Berkeley UNIX are: TCP [RFC793 81], UDP [Postel 80] and IP [RFC791 81]. TCP/IP and UDP/IP provide different kinds of services to the user processes: the former is a virtual circuit based reliable stream with flow control protocol, while the latter is a connectionless datagram protocol. Both TCP and UDP are based on IP for actual data transmissions. IP, as currently implemented, cannot be accessed directly by user processes.

A brief introduction of UNIX sockets is given in Appendix B.

4.2.2 Connectionless broadcast transmission

By using a datagram socket, it is possible to send broadcast packets on many networks connected to the system.

4.2.2.1 Broadcast send in datagram socket

To send a broadcast message, a datagram socket is created:

```
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

The socket is marked as allowing broadcasting,

```
int on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, on, sizeof_on);
```

and at least a port number is bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

After the appropriate ioctl()s have obtained the broadcast or destination address, the sendto() call can be used to send a message.

```
ret_len = sendto(send, mesg, len, 0,
                 (struct sockaddr *) &dest_sin,
                 sizeof(dest_sin));
```

4.2.3 Multicast group and UDP Demultiplexing

We make use of UDP demultiplexing to construct multicast groups. UDP software accepts UDP datagrams from the IP software and demultiplexs them based on the UDP port, as shown in Figure 4.2.

Application programs can define their own port number as long as it does not conflict with the reserved UDP port number used for network services. In our implementation, a unique port number is assigned to each multicast group and all the hosts in the group bind to the same port number. Therefore, during UDP demultiplexing, a datagram packet is sent to all the sites in the group. Figure 4.3 shows two multicast groups active in the system when a datagram packet arrives from IP layer. Assuming the packet is addressed to port 1, after UDP demultiplexing, the packet will be forwarded to every member in Group 1 since they all bind to the port1 but will not be forwarded to Group2 because port1 and port2 are different values.

The reliable multicast communication software is responsible for ensuring that the different multicast groups have different port numbers.

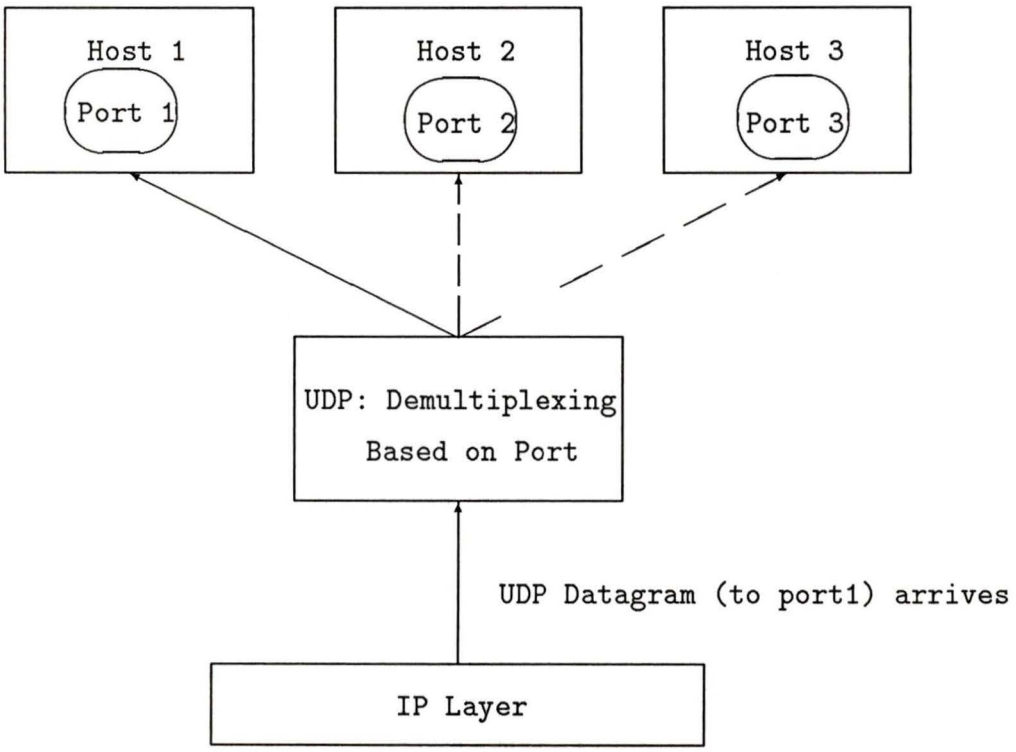


Figure 4.2: **UDP uses the port number to select an appropriate destination for incoming datagram**

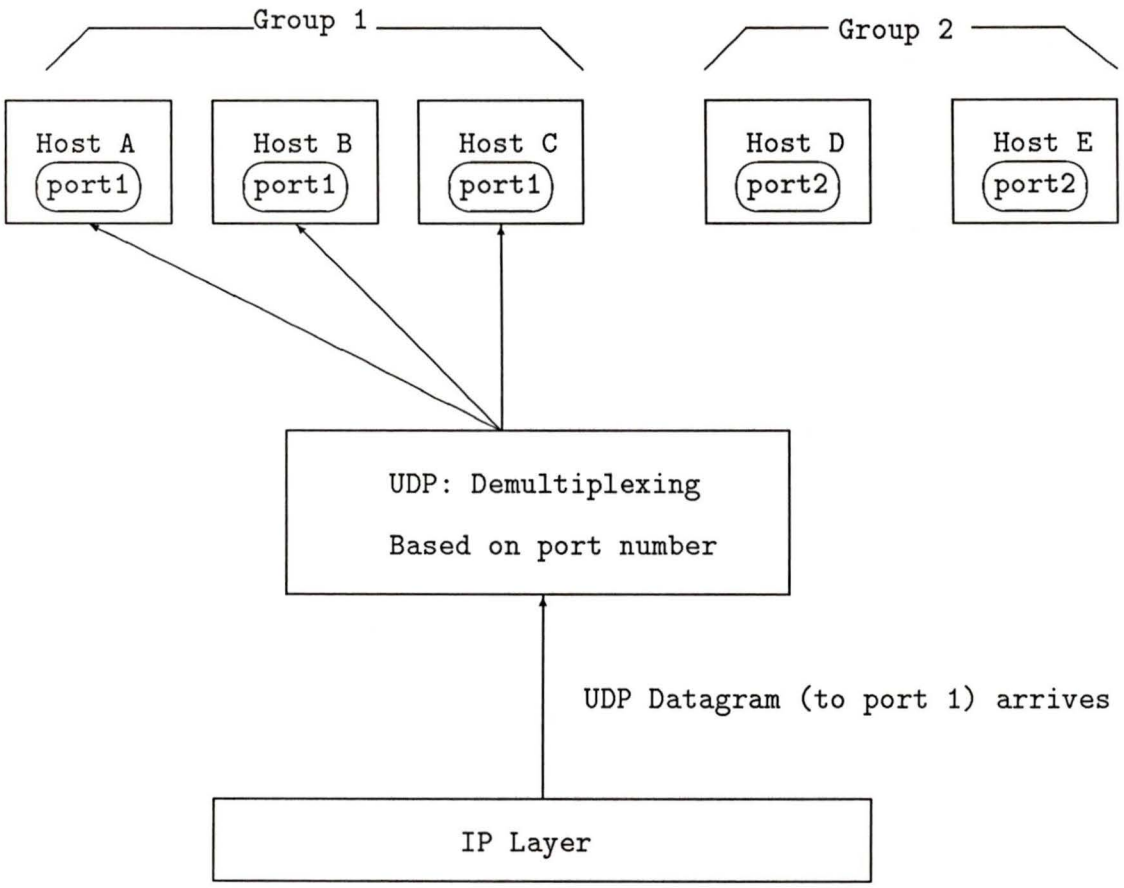


Figure 4.3: **Multicast group receive messages from its unique port number (port1 $\neq$ port2)**

4.2.4 Using a unique port number to construct multicast groups

Port number plays a very important role in the implementation of a reliable multicast protocol. Multicasting is basically achieved by letting a group of processes associate with a common port. Since a port can be "well-known" and shared by processes on different hosts, a broadcast socket can be used for these processes as a common communication channel.

The only problem in our approach is that a process which is not a member of the multicast group, but happens to listen to the common port of that group, may accidentally send or receive messages in the group. We avoid this problem by assigning each multicast group a global unique port number so that messages will not arrive at a wrong destination or be received by a wrong process.

When a process creates a new multicast group, it obtains a unique port number by calling the port number generator. A port number consists of current local time (from *gettimeofday*() system call) and the process identifier. The creator of the group passes the port number to its members. In this way, the multicast communications in different groups will not interfere with each other.

Chapter 5

Multicast Communication in REM

In this chapter, we first give a brief introduction of REM(Remote Execution Manager) and then address the design and implementation details of multicast communication facility in REM.

5.1 REM overview

REM(Remote Execution Manager) [Shoja 88] is a fault-tolerant distributed process manager developed at University of Victoria. The system runs on a local network of Sun 3 diskless workstations connected by 10 Mbps Ethernet. The operating system is SUN OS 4.0 [1]. One of the main objectives of REM was to enable load sharing and parallel processing so that the extra computational bandwidth on remote stations could be utilized. It also provides capabilities

[1]SUN OS 4.0 is based on BSD UNIX 4.3

such as process creation, an interprocess communication protocol, load balancing, fault-tolerance and distributed debugging.

The REM environment consists of two components:

- A set of *daemon processes* which handles intra- and inter-station communications, local/remote process management. These daemon processes are fully distributed without any centralized control.
- An *Application Interface Library* which provides a set of primitives for REM users to access the facility. This library must be linked to the application programs at the compilation time.

Figure 5.1 is an overview of REM architecture.

5.1.1 Terminology

It is necessary to introduce some terminologies which are frequently used in REM. For a complete definition, see [Side 90].

- REM's *daemon process*: the process of a REM running in the background. It provides the means to create, communicate with and terminate remote processes of distributed programs.
- *Application(User) program*: a set of distributed communicating processes that work together to reach a common goal.
- *Application(User) process*: a process of the application program. Each application process uses the routines in REM's Application Interface Library to communicate with and control other processes of the application program.

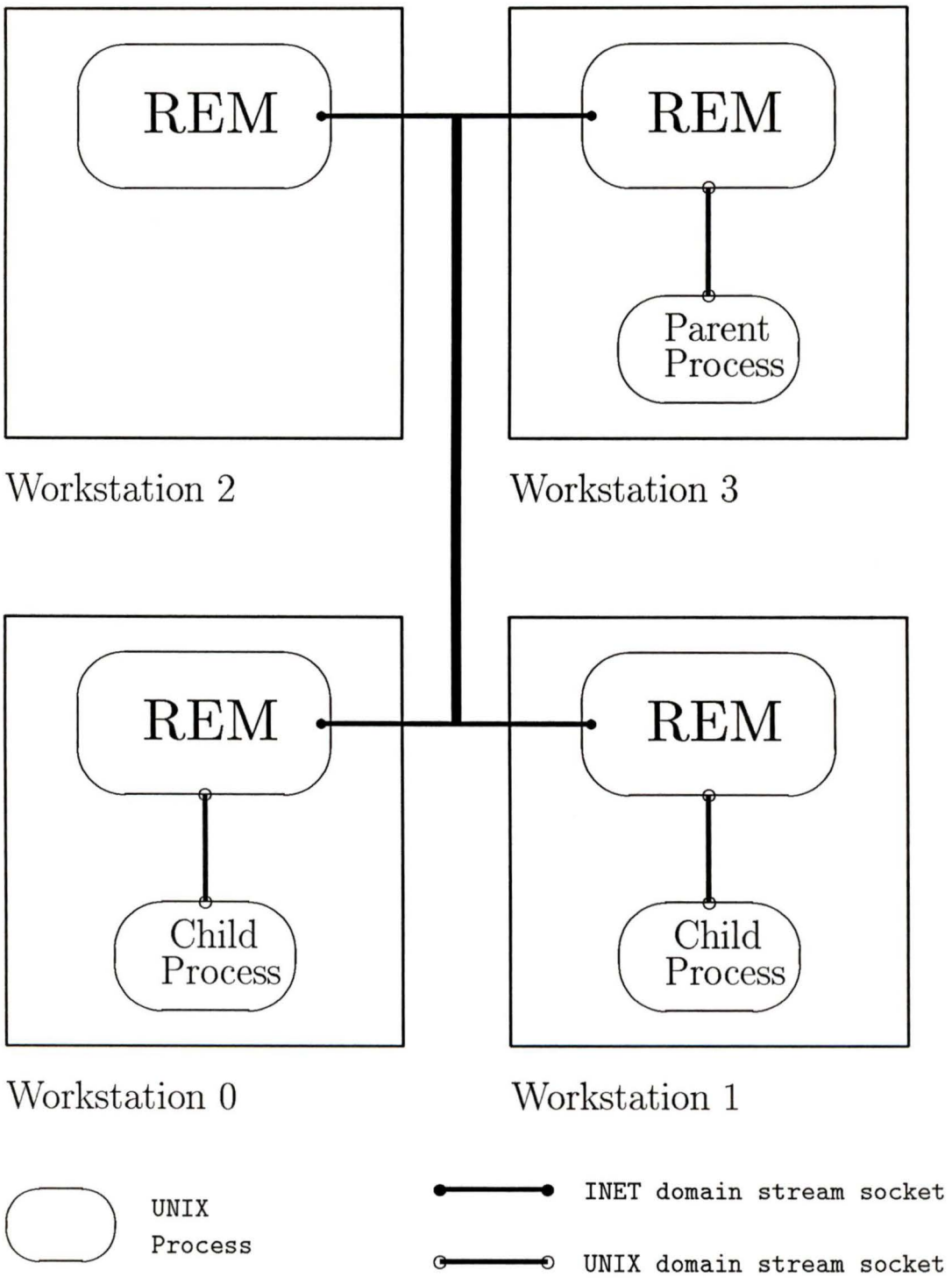


Figure 5.1: **An overview of REM architecture**

- *Parent (application) process*: the first process invoked when an application program executes. There exists at most one parent application process per application program and all other application processes are offsprings of this process.
- *Child process*: a remote process created by the parent application process through the facility supplied by REM. There may exist many child application processes per application program.
- *Suite*: a collection of workstations each containing a daemon process. Each daemon process has knowledge of all other daemon processes in the suite.
- *Host workstation*: the workstation in a suite where the application program is invoked and the only workstation the parent process can reside on.
- *Remote workstation*: all other workstations in a suite and where the child processes reside.

5.1.2 REM process management and communication methodology

In a REM environment, REM daemon processes handle process creation and termination. All communications for each station is handled by the station's REM daemon process.

5.1.2.1 Process management

In REM model, the concept of **suite** is very important. A suite is a collection of REM daemon processes. Each daemon in a suite knows all the other daemon processes in the same suite but has no knowledge about any daemon in other

suites. There can be more than one active suite on the network at any time, and an individual workstation may have more than one suite running on it as shown in Figure 5.2.

A remote child process is initiated by a parent process by passing the name of the child application program to the local REM daemon process. This REM daemon process then passes the name to a remote REM machine for execution. REM user interface library provides access routines for user processes to communicate with REM daemons.

Once the child processes are successfully created, the parent process and child processes can communicate by sending the message to their local REM daemon. REM daemon will transmit/receive the messages through inter-process communication facility.

Upon finishing computation, both parent and child process issue a termination request to clear up the records kept in REM.

REM's user interface provides a set of primitives as follows:

initialize_dist(): contact REM daemon process.

create_dist() : send a request to REM daemon to initiate the child process creation.

send_dist() : send a message to parent/child process.

us_recv_dist() : receive a message from parent/child process.

terminate_dist() : disconnect REM daemon process.

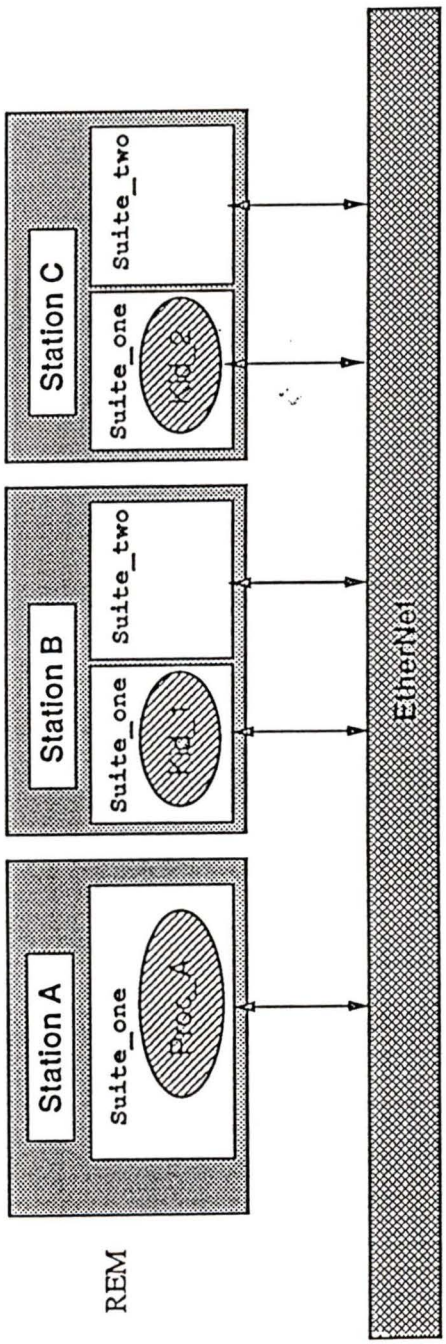


Figure 5.2: Process in REM model

5.1.2.2 Inter-process communication

In REM model, all communications are parent-to-child or child-to-parent, child-to-child communication is not provided. Parent as well as remote child processes, communicate with their respective local REM daemon processes through UNIX domain stream sockets. REM daemon processes on different machines communicate with each other using INTERNET domain stream socket.

Figure 5.3 shows the interprocess communication modules of REM.

5.2 Implementation of multicast communication in REM

5.2.1 Overview

This section describes the implementation details of the proposed reliable multicast communication protocol in the REM environment. Originally, REM only provided one-to-one interprocess communication. A user had to call the unicast primitives several times to send a message to a set of processes. With the multicast communication facility that we have implemented in REM, the user only needs to issue the send request once thus reducing the communication time.

For each REM daemon process, a pair of internet datagram socket is created when entering multicasting mode. One for sending out multicast messages and one for receiving messages multicast by other hosts. In multicast mode, the inter-station communication in REM is through connectionless datagram instead of the original connection-oriented stream type of communication. The overview of multicast communication in REM is presented in Figure 5.4.

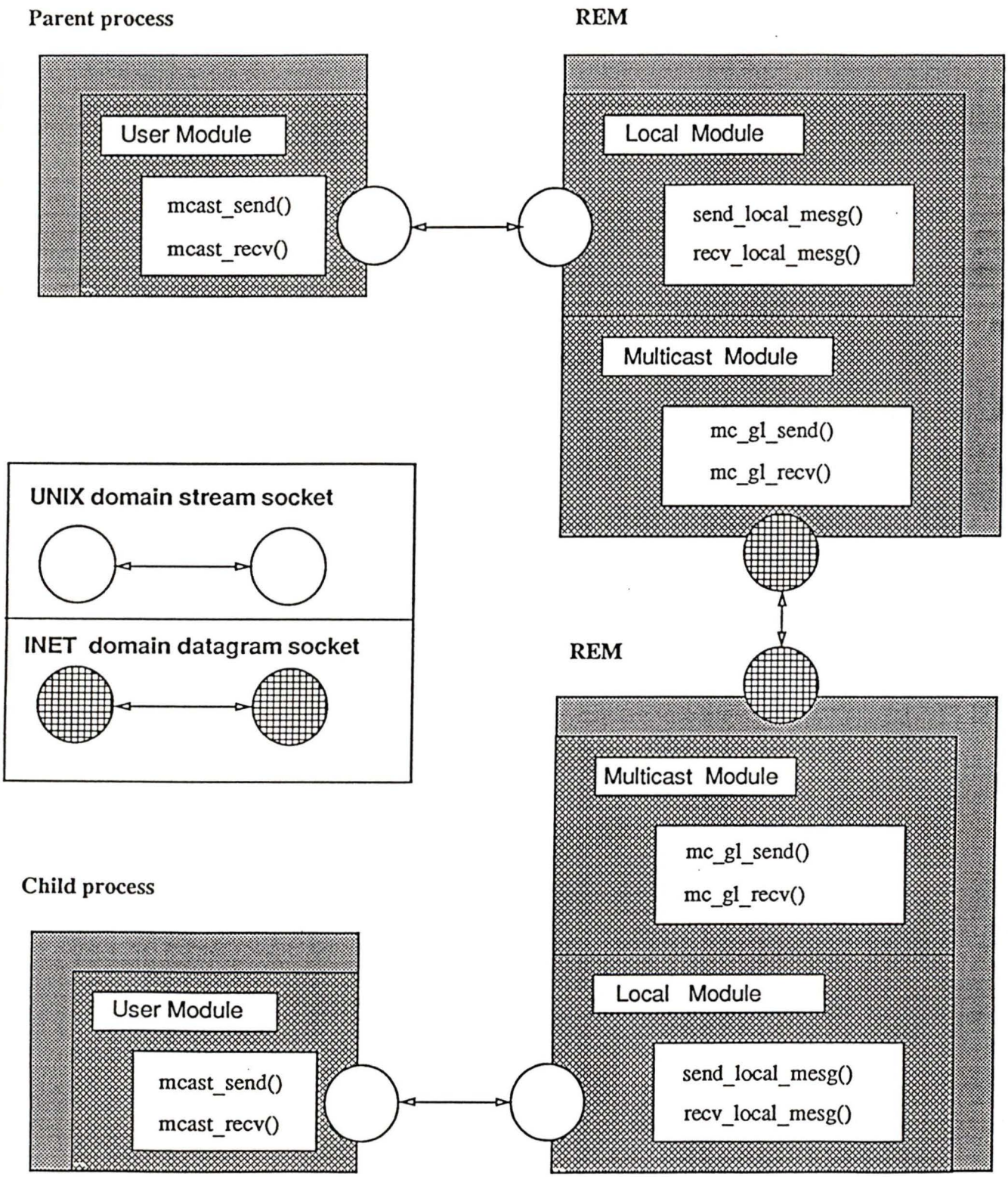


Figure 5.3: Interprocess communication modules of REM

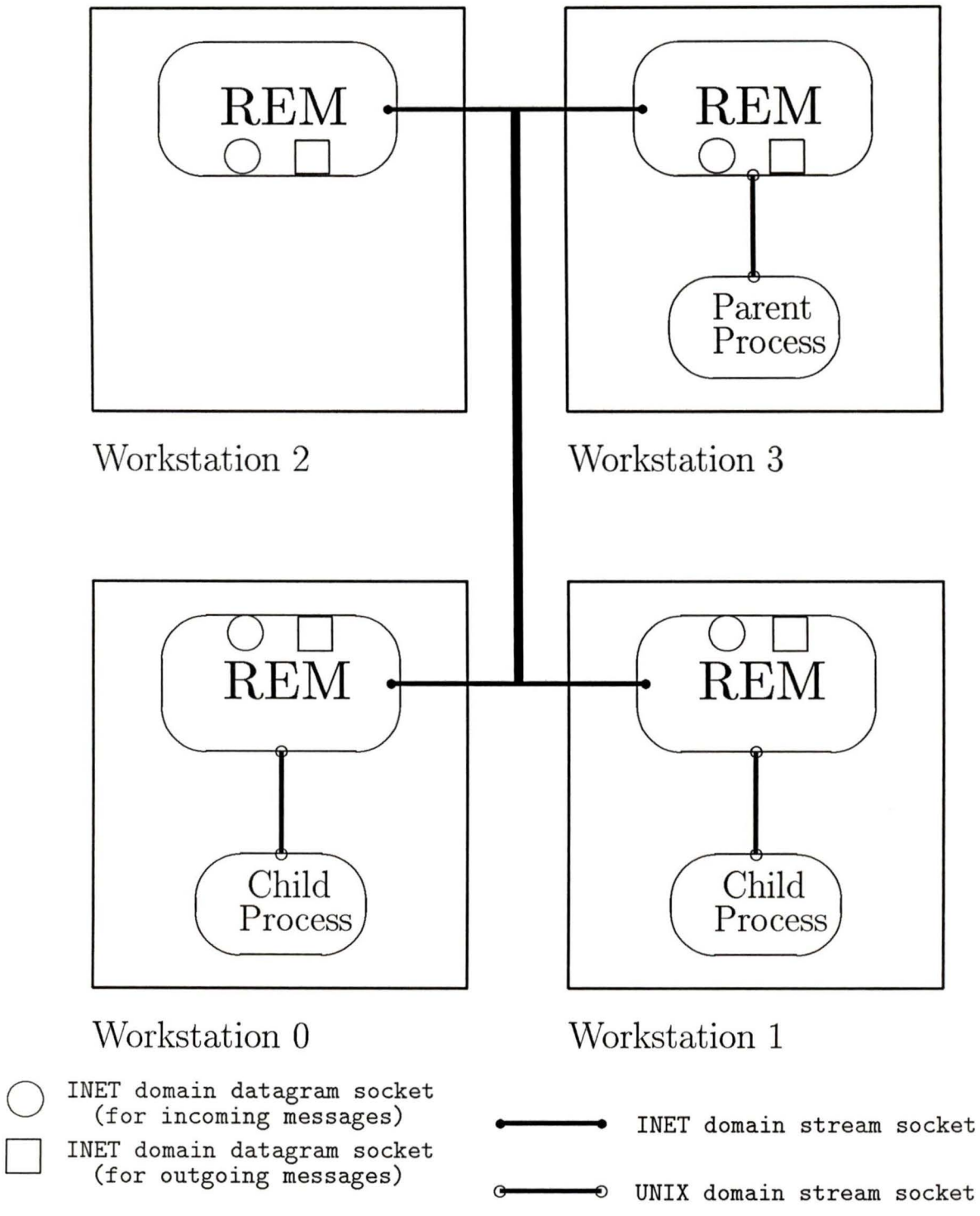


Figure 5.4: **REM communication(with multicast) architecture**

5.2.2 Design considerations

5.2.2.1 Multicast group management

Due to the restriction of process management and communication architecture in REM, the implementation of the reliable multicast protocol proposed in Chapter 3 had to be modified to suit the REM environment. For example, *mjoin* primitive is not implemented in REM because a child process is created by a parent process through REM. A process cannot voluntarily be a child process of a certain parent process. As a result, a process cannot dynamically join a multicast group.

In the REM environment, a multicast group consists of a parent process and the child processes the parent created. At the time a child process is created, it automatically becomes a member of the group. Since child-to-child communication is not currently supported in REM, a child has no idea of the other children's whereabouts. This makes maintaining the *group view* difficult. One approach to solve the problem is for the parent to keep the group member list and broadcast to its children whenever there is a change. This is rather costly due to the extra communication overhead involved. In our implementation, we make use of the fact that every REM daemon has the knowledge of the locations of all the other REM daemons in the same *suite*. Therefore, we use suite to form a multicast group and assign a unique *common port number* to it.

When a station crashes, the information becomes known to the suite and the process residing on the failed station will be removed from the group membership list.

5.2.2.2 Synchronous type of communication

In our reliable multicast protocol, a receiver has to wait for all the acknowledgements sent by the other receivers in the same group. If some receivers take a long time to process the sender's message and subsequently delay the broadcasting of the acknowledgements, the local REM daemon cannot unblock the sender. However, if the user wishes atomicity of message transmission, there is no other alternative but waiting until the consistent status is reached.

5.2.2.3 Modularized design

When implementing multicast communication mechanism in REM, we tried to follow the modularized design and make as few changes to the core REM software as possible. Also, the user interface of multicast communication facility was made similar to the original REM user interface. The message exchanges for ensuring reliability properties are handled within REM daemon processes and are transparent to the application programs.

5.2.2.4 Modification to REM

In order to integrate the multicast communication facility into REM, some modifications had to be made to the original REM software.

(a) New message types

The following new message types were added to REM's original message types.

```
INITIAL_S_MCAST /* initialization for multicast send */
INITIAL_R_MCAST /* initialization for multicast receive */
```

```
MC_CREATE_RELIA /* strong reliable multicast */
MC_CREATE_UNREL /* weak reliable multicast */

MC_SEND_MESG  /* multicast send message */
MC_RECV_MESG  /* multicast receive message */

MC_S_TERMINATE  /* termination for multicast send */
MC_R_TERMINATE  /* termination for multicast receive */
```

(b) User interface module

A set of primitives for multicast communication were added to the REM's user interface module and the appropriate routines were provided in REM user interface library. The syntax of these primitives are very similar to the REM unicast communication primitives so that the users who are already familiar with REM will have no difficult in using them.

(c) Local process management module

The local process management module in REM processes the messages passed to the local REM daemon from user processes. When a message arrives, the message type in the message header is checked. If the current message is a multicast communication request (creation, send, receive, etc.), the corresponding routines in multicast communication module are called.

5.2.3 Implementation details

The main consideration during our implementation was to minimize the modification to the original REM structure and to achieve high efficiency as well. The multicast communication facility resides rather independently in REM. Only when a user issues a multicast communication request then REM switches its inter-station communication mode from unicast to multicast.

There are mainly three modules: *User Interface Module*, *Multicast Communication Module* and *Reliable Multicast Protocol Module.*

5.2.3.1 User interface module

Similar to the REM's user interface, the user interface of multicast communication provides a set of functions for users. User requests with the embedded message types are passed to the local REM through UNIX domain socket.

(a) Initialization

initiate_s_mcast(group_size);

The sender calls this primitive to initialize the environment for sending multicast messages. The functions include setting the flag to indicate multicast mode, initializing local variables and queues and creating a multicast send socket. The *group_size* is the number of processes participating in the multicast communication. This number cannot be larger than the number of machines on which REM is running.

initiate_r_mcast(reliability_flag);

The receivers call this primitive to initialize the environment for receiving multicast messages. The functions include setting the flag to indicate multicast mode, initializing local variables and queues and creating multicast receive sockets. The *reliability_flag* is set to 1 when the user wants to use the reliable multicast communication facility (when the sender uses **mcreate_relia** to create child processes). It is set to 0 to tell the local REM that the

user wants to use the multicast facility with weak reliability (when the sender uses **mcreate_unrel** to create child processes).

mcreate_unrel(child_process_name)

This primitive is used for creating child process(es) which will use simple multicast communication protocol for interprocess communication.

mcreate_relia(child_process_name)

This primitive is used for creating child process(es) which will use reliable multicast communication protocol for interprocess communication.

(b) Communication

msend(ppid, mesg, mesg_len)

This function passes the user message and the message length with an embedded message type, indicating this is a multicast message, to local REM. Upon receiving the message, REM calls the routines in the *multicast communication module* to broadcast the message to remote machines. If the current communication mode is simple multicast, the local REM daemon will unblock the sender after the call returns by sending a local ACK message through UNIX domain socket. If the current communication is reliable multicast, the sender will remain blocked after invoking *msend* until the REM daemon ensures the message has been received by all the remote REM daemons

mrecv(ppid, mesg_buf, buf_size)

This function blocks the user process to wait for the message. It

passes the message buffer address and the buffer size to the local REM. For simple multicasting, the local REM daemon delivers the message to the user as soon as it is received. For reliable multicasting, however, the message is delivered to the user only after the local REM is convinced that the same message has been successfully received by all the recipients.

(c) Termination

mremove()

Sender calls this primitive to inform local REM to terminate current multicast communication. The multicast group will be removed. The multicast sockets will be closed.

mleave()

Receiver calls this primitive to inform local REM that it is leaving the current multicast group. The multicast sockets will be closed and the receiver process will not receive any further messages from this group..

5.2.3.2 Multicast communication module

This module handles the inter-station communication between REM machines. The main functions are sending and receiving multicast messages among machines.

For some applications, atomicity and global ordering are not required. We provide a simple multicast protocol in REM for the convenience of the users who do not need the strong reliability. User messages are sent immediately

when they arrive at local REM and the messages from remote machines are delivered to the user as soon as they are received.

(a) Initialization

When REM receives the multicast initialization request from user, it creates a pair of Internet domain sockets, one for sending multicast messages and one for receiving multicast messages. Both are connectionless datagram sockets. These sockets will be closed when user issues a termination request.

For example, suppose the *common port number* is 98765, and consider the following scenario, showing how a process P_a sends multicast messages to P_x(x is any of the receivers in the group which P_a belongs to).

At the P_a:

i. Create an INTERNET datagram socket with UDP protocol.

ii. Get which net we are on.

iii. Load up the common port address.

The corresponding C code is as follows:

```
if ((send_sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
        perror("sockinit (socket)");
        gl_terminate(1);
   }

   /*
    * Get which net we are on and then load up the common port address
    */
   if ((which_net = mc_get_bcast_net(in, send_sock, inbuf)) <= 0) {
        perror("sockinit (no nets)");
        gl_terminate(1);
   }

   bzero((char *) &mcast_to_sin, sizeof(mcast_to_sin));
```

```
mcast_to_sin.sin_family = AF_INET;
mcast_to_sin.sin_addr.s_addr = htonl(INADDR_ANY);
mcast_to_sin.sin_addr = in[0];
mcast_to_sin.sin_port = htons(98765);
```

At the P_x:

i. Create an INTERNET datagram socket with UDP protocol.

ii. Load up the common port address.

iii. Bind a name to the socket.

The corresponding C code is as follows:

```
if ((recv_sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
     perror("sockinit (socket)");
     gl_terminate(1);
}
bzero((char *) &mcast_from_sin, sizeof(mcast_from_sin));
mcast_from_sin.sin_family = AF_INET;
mcast_from_sin.sin_addr.s_addr = htonl(INADDR_ANY);
mcast_from_sin.sin_port = htons(98765);

if ((bind(recv_sock, (struct sockaddr *) &mcast_from_sin, len)) < 0) {
     perror("mc_sock_recv_init (bind)");
     gl_terminate(1);
}
```

(b) Communication

After successful socket creation, we can send/receive multicast messages. While using INTERNET datagram sockets, messages are sent by calling *sendto* system call and are received by using *recvfrom* system call. *sendto* is an unblock function while *recvfrom* is a block type system call which would block the caller until the message arrives at the socket. Before actually reading any message, the socket is *peeked* to see whether the there is any data available on the socket. If the *peeking* successfully returns(with

the length of the data), a *recvfrom* system call is issued to read the message.

```
Sending...
        ret_len = sendto(mc_send_fd, mc_out_mesg, mc_out_len, 0,
                         (struct sockaddr *) &mcast_to_sin,
                         sizeof(mcast_to_sin));

Peeking...
        mc_ret_val = recvfrom(mc_recv_fd,
                         &mc_peek_len,
                         sizeof(u_long),
                         MSG_PEEK,
                         (struct sockaddr *) &mcast_from_sin,
                         &mc_tmp_fromlen);
Receiving...
        mc_ret_val = recvfrom(mc_recv_fd,
                         mc_in_mesg,
                         sizeof(typ_msg), 0,
                         (struct sockaddr *)&mcast_from_sin,
                         &mc_tmp_fromlen);
```

For weak reliable multicast communication, message transmission and reception are handled in this module.

5.2.3.3 Reliable Multicast Protocol Module

This module deals with reliability issues in the protocol, such as updating local *current status*, timestamping, sending acknowledgements and retransmission messages.

(a) New message types

Some message types are defined for the use of the reliable protocol.

```
#define MCAST_MESG   11 /* multicast send message */
#define MCAST_ACK    12 /* multicast acknowledgement message */
#define MCAST_CFM    13 /* multicast conformation message */
#define MCAST_REX    14 /* multicast retransmission message */
```

(b) Initialization

Besides the work similar to the initilization in *multicast communication module*, several local variables for recording multicast status are also initialized, such as CS vector for current status, TS vector for timestamp and SEQ vector for keeping track of message sequence.

(c) Send

When a REM daemon receives a message to send a multicast message and the current communication mode is reliable communication, it assigns the message a $MCAST\_MESG$ message type and sends the message along with the local CS and TS vector and sequence number. It then sets the timer and waits for acknowledgements. When a message arrives, the message type in the header is checked. If the message is a acknowledgement from one of the members, the REM daemon records its arrival by updating the local CS vector. The element corresponding to the source will be set to 1. If the local REM fails to collect all the acknowledgements within a certain time interval, it retransmits the multicast message with $MCAST\_REX$ message type in the header.

Duplicate messages are detected here. In the original one-to-one IPC in REM, since the connection-oriented stream socket with TCP/IP protocol is used, the receipients will not receive duplicate messages. However, in mulitcast communication, where messages are retransmitted in datagram fashion using UDP/IP protocol, receipients may receive duplicate messages. Also, the sender will receive a copy of the message it just sent out, which is discarded.

(d) Receive

When a REM daemon receives a $MCAST\_MESG$ type of message, it

constructs an acknowledgement message with $MCAST\_ACK$ and its local CS and TS vectors. It then broadcasts the message and enters a blocking state to wait for the other acknowledgements. The REM daemon also updates the local vectors based on the remote vetor values. If the message is a retransmission of a previous message which has been received then it is discarded, otherwise it is keeped and acknowledged.

Upon collecting all the acknowledgements from members. The local REM unblocks the user process by delivering the message.

Chapter 6

Results and Evaluation

In this chapter, we present the results of the benchmarks used to evaluate the performance of the multicast communication, with weak and strong reliability, in REM.

We do not intend to compare the performance of our protocol with other multicast communication protocols in terms of execution time for the following reasons:

- Different protocol performances are difficult to compare. They are often performed with different protocol parameters (the layer at which they are implemented, the underlying protocol used, etc.), on different computers and networks.
- The implementation environment – REM is implemented on top of UNIX system at application layer and our protocol is implemented in REM. As a result, the performance results are uncomparable with those protocols built in the transport layer or within the system kernel.

Therefore, we consider that the comparison of the results within REM is more meaningful.

The objectives of these tests were twofold. First, to compare the efficiency of multicast communication to that of unicast communication. Second, to determine the overheads associated with the reliable multicast communication protocol.

6.1 Measurement environments

There are many environmental factors which may affect performance, such as configuration, parameters of the machine and underlying network, CPU and network load, etc.

Our measurement environment consisted of:

Machine: Fourteen SUN 3/60 (diskless) workstations connected to two Sun 3 servers.

OS: Standard SUN OS 4.0.

Memory: 4 Mbyte.

Network: Ethernet (10 Mb/s).

Our data were obtained under the following conditions:

Network load: Low load factor.

Machine load: Machines run in single-user mode.

REM load: Only one user is running REM on any of the machines.

To ensure consistency with respect to the measurement environments, all the data were collected during off-hours (midnight to early morning) when the system and network were lightly loaded. Also, the times shown in the results are the averages for 10 runs.

We used three benchmarks to test the protocol's speed with respect to: 1) parallel computation; 2) sending of a message; 3) sending of a message and waiting for an immediate reply. These benchmarks also serve to demonstrate how to use the multicast facility in REM. The *C* source code for each of the benchmarks is included in Appendix D.

Before further discussion, we define some terminologies used in our tests:

- *completion time:* the elapsed time from sending the task to the child processes until receiving replies from all the children.
- *send time:* the elapsed time during which a sender remains blocked after invoking a multicast send primitive.
- *computation time:* the time a child process spends in accomplishing a task.
- *response time:* the time it takes to send a message to a remote machine and get a response message back. This is used only when the message is short and no computation required at the child machine. *Response time* is also called round trip time. When there are more than one remote machine, this response time is the elapsed time from sending the message until receiving all the responses back from every remote machine.

6.2 Distributed CPU intensive task

This is one of the REM's parallel execution benchmark. We use it to compare REM unicast communication and multicast communication. Since we are mainly concerned with the performance of the multicast communication facility, we send the whole task to every child process instead of dividing it into several small tasks and distributing them among child processes.

This test creates child processes and distributes the task, a loop consisting of long divides, to the children. The cpu loop is a long divide which is executed one thousand times. The child processes receive a message indicating how many loops to perform. When run with zero children, all loops are performed locally, when run with one or more children, the number of loops are sent to every child and performed on remote machines. Our tests run on up to 8 children.

Table 6.1 shows the completion time of the task execution using REM unicast, weak reliable multicast and strong reliable multicast communication to pass the number of loops to the child processes.

From the data we can see that there is a significant speedup of completion time when using multicast communication facility, especially when the number of the receivers is increased as shown in Table 6.2.

Table 6.3 is the comparison of the communication overhead when using reliable versus simple multicast.

6.3 Message passing – send time

The message passing program creates a set of child processes and sends them a number of messages. The program was tested with fixed message size (900

Number of Loops	Number of Children	Completion time (real time, sec) Unicast	Completion time (real time, sec) Weak Multicast	Completion time (real time, sec) Strong Multicast
50	0	4.252	-	-
50	1	5.270	5.274	5.311
50	2	5.844	5.280	5.328
50	3	5.884	5.316	5.329
50	4	6.070	5.320	5.360
50	5	6.390	5.330	5.410
50	6	6.810	5.400	5.608
50	7	7.964	5.475	5.764
50	8	8.592	5.540	6.120

Table 6.1: **The results for completion time in parallel execution program**

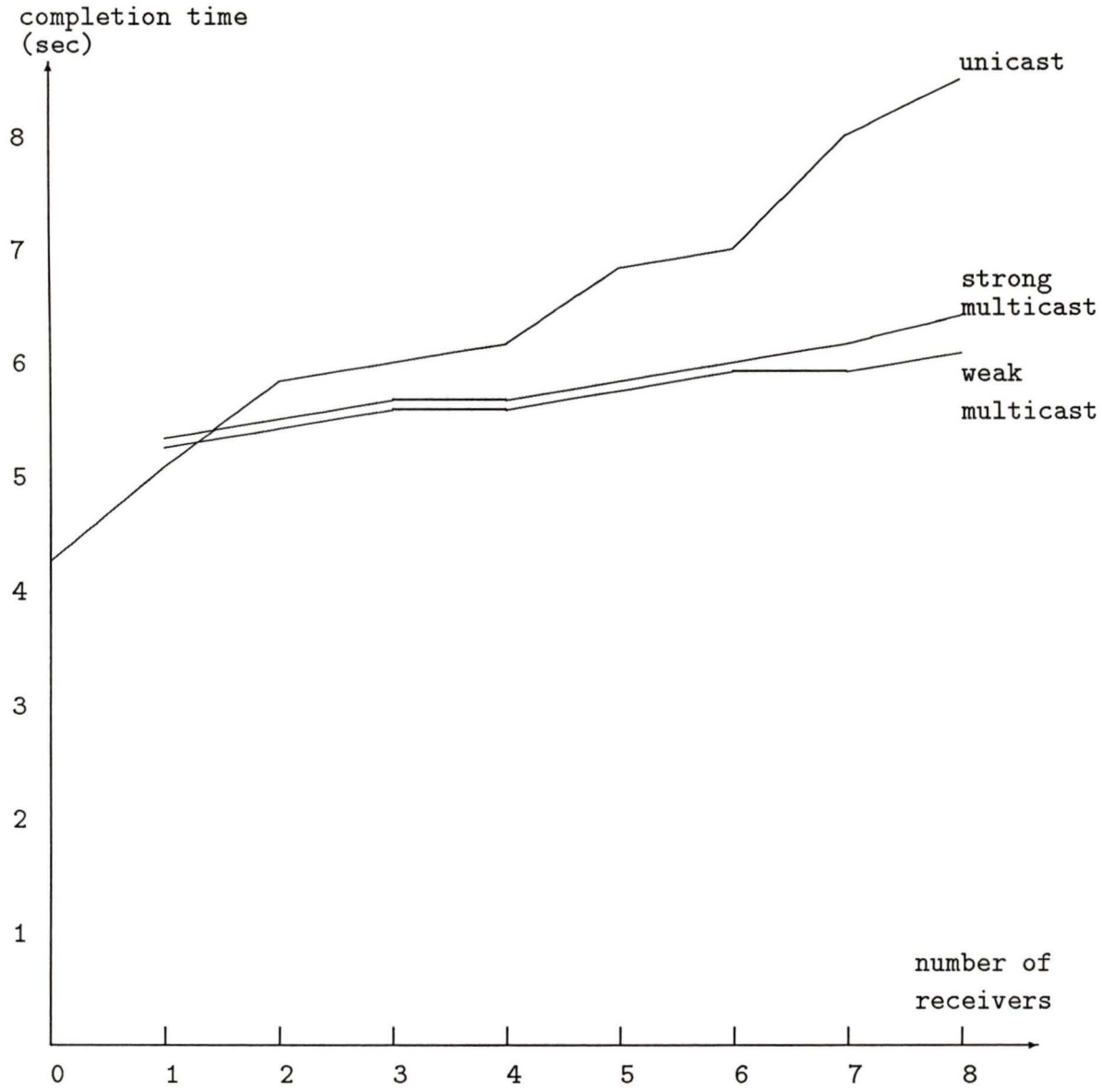


Figure 6.1: **performance in completion time**

Number of Loops	Number of Children	Completion time *real time, sec* Unicast	Completion time *real time, sec* Weak Multicast	Speedup ($T_{unicast} - T_{multicast}$ / $T_{unicast}$)
50	0	4.252	-	-
50	1	5.270	5.274	0.08%
50	2	5.844	5.280	9.65%
50	3	5.884	5.316	9.65%
50	4	6.070	5.320	12.36%
50	5	6.390	5.330	15.02%
50	6	6.810	5.400	20.70%
50	7	7.964	5.475	31.25%
50	8	8.592	5.540	35.52%

Table 6.2: **Unicast vs. simple multicast: performance speedup in completion time**

Number of Loops	Number of Children	Completion time (real time, sec) Weak Multicast	Completion time (real time, sec) Strong Multicast	Overhead
50	1	5.274	5.311	0.07%
50	2	5.280	5.328	0.09%
50	3	5.316	5.329	0.24%
50	4	5.320	5.360	0.75%
50	5	5.330	5.410	1.48%
50	6	5.400	5.608	3.71%
50	7	5.475	5.764	5.01%
50	8	5.540	6.120	9.48%

Table 6.3: **Reliable vs. simple multicast: protocol overhead in completion time**

Number of Children	Number of Message	Message Size (bytes)	Send time (real time, sec) **Unicast**	Send time (real time, sec) **Weak Mcast**	Send time (real time, sec) **Strong Mcast**
1	1	900	0.014	0.020	0.028
2	1	900	0.024	0.020	0.039
3	1	900	0.036	0.020	0.045
4	1	900	0.040	0.020	0.054
5	1	900	0.054	0.020	0.059
6	1	900	0.062	0.020	0.060
7	1	900	0.074	0.020	0.067
8	1	900	0.086	0.020	0.071

Table 6.4: **The results for *send* time in message passing program**

bytes) but on a different number of workstations (up to 9). The *send time* is shown in Table 6.4.

We have several observations from these figures:

- The elapsed time of weak multicast send primitive is the shortest, as expected.
- The elapsed time of strong multicast primitive is rather long compared with both unicast and weak multicast. This is because the strong multicast primitive provides a synchronous type of communication, that is, sender remains blocked until the underlying system(REM) gets confirmation messages from all the receivers showing the message was successfully delivered. In unicast and weak multicast, however, the sender is unblocked

as soon as REM receives the message and calls the interprocess communication routines.

- In weak multicast, the increase in the elapsed time for additional remote members is not very significant, in fact, they are almost constant. This behaviour is understandable, because the underlying network was a broadcast network and the time to transmit a multicast message to one remote site or to multiple remote sites is the same.

6.4 Message passing – response time

The response times are shown in Table 6.5. The performance improvement in response time is shown in Table 6.6. Table 6.7 shows the overhead of reliable protocol. The time we have measured here is the elapsed time from when the sender issues a send primitive until it gets all the replies from all the receivers. Unicast is used as the mechanism of getting the replies.

6.5 Summary

In this chapter we have presented three benchmarks to measure the performance of multicast communication facility we have implemented for REM. From these results we can draw the following conclusions:

- Efficiency

  The results show that user programs gain a significant improvement in communication time by using multicast communication facility to send messages to a set of remote machines. Using unicast, the processing and

Number of Children	Number of Message	Message Size (bytes)	Response time (real time, sec) Unicast	Response time (real time, sec) Weak Mcast	Response time (real time, sec) Strong Mcast
1	1	900	0.184	0.239	0.256
2	1	900	0.240	0.256	0.280
3	1	900	0.334	0.264	0.330
4	1	900	0.540	0.352	0.450
5	1	900	0.626	0.378	0.529
6	1	900	0.732	0.486	0.679
7	1	900	0.856	0.532	0.802
8	1	900	0.912	0.590	0.892

Table 6.5: **The results for response time in message passing program**

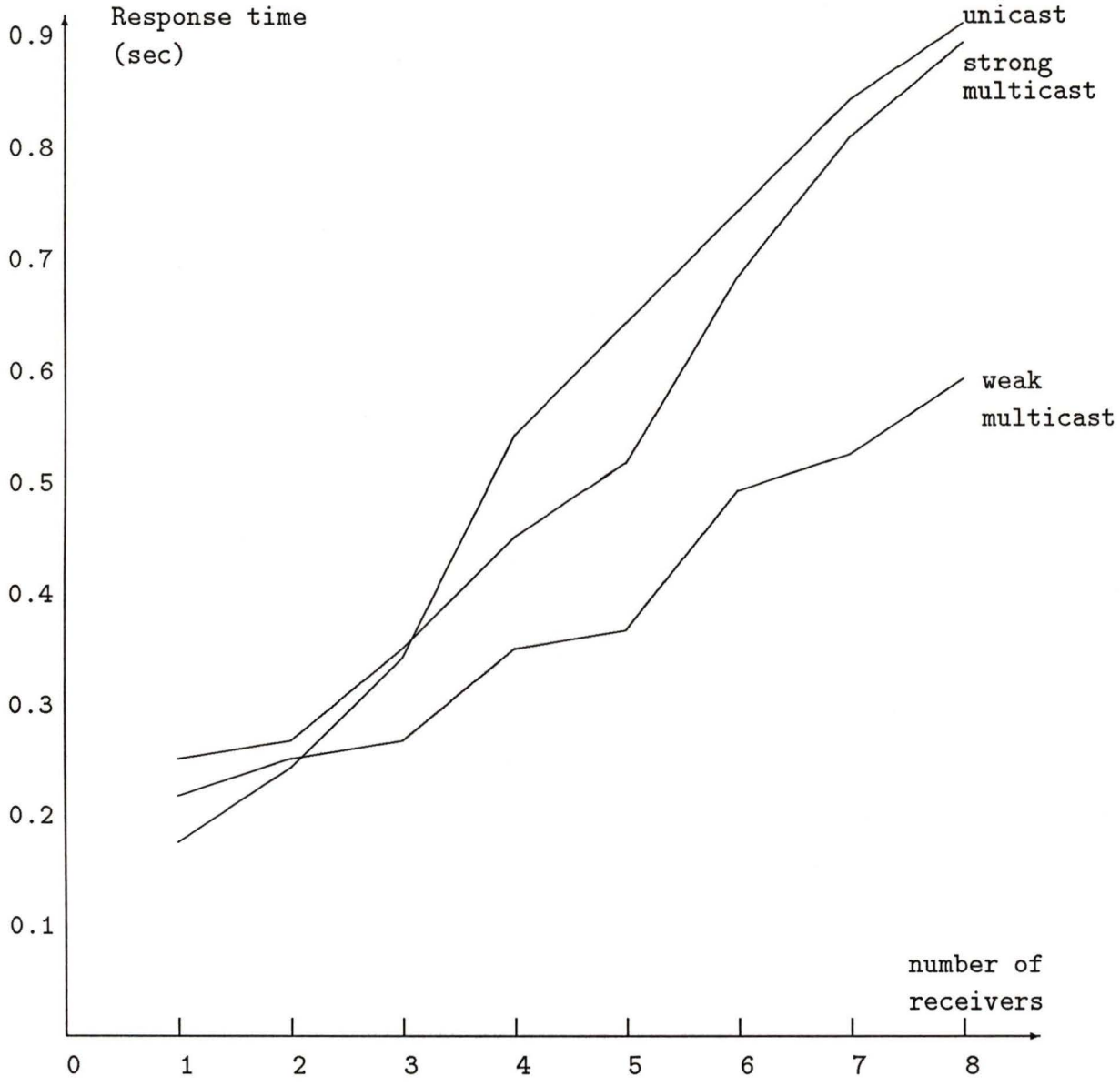


Figure 6.2: **performance in response time**

Number of Children	Number of Mesg.	Message Size (bytes)	Response time (real time,sec) Unicast	Response time (real time,sec) Weak Mcast	Speedup ($T_{unicast} - T_{multicast}$ / $T_{unicast}$)
1	1	900	0.184	0.239	-
2	1	900	0.240	0.256	-
3	1	900	0.334	0.264	20.96%
4	1	900	0.540	0.352	38.81%
5	1	900	0.626	0.378	39.62%
6	1	900	0.732	0.486	33.60%
7	1	900	0.856	0.532	37.85%
8	1	900	0.912	0.590	35.31%

Table 6.6: **Unicast vs. simple multicast: performance speedup in response time**

Number of Children	Number of Message	Message Size (bytes)	Response time (real time, sec) Weak Mcast	Response time (real time, sec) Strong Mcast	Overhead
1	1	900	0.239	0.256	6.64%
2	1	900	0.256	0.280	8.57%
3	1	900	0.264	0.330	20.00%
4	1	900	0.352	0.450	21.77%
5	1	900	0.378	0.529	28.54%
6	1	900	0.486	0.679	28.42%
7	1	900	0.532	0.802	33.66%
8	1	900	0.590	0.892	33.85%

Table 6.7: **Reliable vs. simple multicast: protocol overhead in response time**

network transmission costs of communicating within a process group rise linearly with the number of processes in the group. Using multicast, the cost increasing speed is much lower than unicast.

- The cost and performance depend on the degree of reliability desired.

  We discussed the overhead of reliable communication and noticed this protocol is more costly than simple multicast communication protocol. This is because we provide a strong reliability for application programs. For users who do not need such a high degree of reliability, this protocol is not recommanded.

- Reliable multicast overhead

  In our reliable multicast protocol, as the number of processes in a group increases, the time spent on achieving consistency in the group becomes significant. Also, a process sending a message to a group may experience congestion at the network interface as many acknowledgement packets arrive more or less simultaneously from other processes in the group.

- Compatibility with REM

  The multicast communication facilities co-exist very well with REM unicast communication. A user can use either type of communication for his/her applications, or use two of them alternately, as long as the communication environments are created before the communication requests are issued.

Chapter 7

Conclusion

Multicast communication is a useful tool for parallel computation and many distributed applications. In this thesis we have introduced a high level protocol for reliable multicast communication in distributed environment. The design and implementation details of the protocol were also described. The multicast communication mechanism satisfies the three reliability properties: full delivery, atomicity and global ordering.

7.1 Contribution

Our main contribution has been to propose and design a reliable multicast protocol and implement it as part of a practical distributed and parallel computing environment. The protocol we have proposed is fully decentralized and does not require clock synchronization.

The protocol is implemented in the Remote Execution Manager (REM). The implementation is entirely on top of the operating system and does not require

any changes to the kernel, therefore, the portability of REM is retained. Some modifications to REM core were made in order to integrate the multicast facility, but the original REM features were not affected.

Unlike in some other protocols that the underlying system or network provides reliable message transmission, our protocol is based on a unreliable datagram transmission. We introduced two vectors used in the protocol and explained how they were used for synchronization and consistency control.

We also introduced a set of high level multicast communication primitives and implemented the primitives in REM.

Some performance measurements were made on the multicast communication protocols(with weak and strong reliability) implemented in REM. The benchmark results show that the overhead of reliable multicast communication is significant. This is due to the fact that every reliable multicast send operation is synchronous. To ensure atomicity, a user process is blocked from sending the next message until the underlying system has delivered the current message to every non-faulty member. The results also show that the simple multicast communication is an efficient and useful facility when the user does not require a high degree of reliability.

7.2 Future works

There are two main criteria for a communication protocol, one is the degree of reliability it provides and the other is efficiency. The definition of *reliability* in our reliable multicast communication protocol is rigorous. However, some improvement is needed for the efficiency of the protocol. If the underlying

system supports group communication, such as V system or T-shoshin, one can expect some speedup at protocol level since the overhead for group management would be greatly reduced.

A theoretical proof of the correctness of our reliable multicast protocol is an interesting topic for future research.

We have assumed *fail-stop* processors in our protocol . One will need to modify the protocol in order to remove this assumption. The protocol can then tolerate *Byzantine* failure when a faulty processor can send corrupted messages to the others.

Also, it would be useful to implement dynamic join and leave primitives in REM.

Bibliography

[Ahamad 85] Ahamad, M., Bernstein, A., "Multicast Communication in UNIX 4.2BSD," *Proceedings of IEEE the 5th International Conference on Distributed Computing Systems*, May. 1985, pp. 80-87.

[Birman 87] Birman, K.P., Joseph, T.A., "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, Vol. 5, No. 1, Feb. 1987, pp. 47-76.

[Birrell 84] Birrell, A.D., Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, Aug. 1984, pp. 251-273.

[Chang 84] Chang, J-M., Maxemchuk, N.F., "Reliable Broadcast Protocol," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, Aug. 1984, pp. 251-273.

[Cheriton 84] Cheriton, D.R., Zwaenepoel, W., "One-to-Many Interprocess Communication in the V-System," *ACM SIGCOMM'84 Symposium*, 1984.

[Cheriton 85a] Cheriton, D.R., Zwaenepoel, W., "Distributed Process Groups in the V Kernel," *ACM Transactions on Computer*

Systems, Vol. 3, No. 2, May. 1985, pp. 77-107.

[Cheriton 85b] Cheriton, D.R., Deering, S.E., "Host Groups: A Multicast Extension for Datagram Internetworks," *Proc. of the 9th Data Communication Symposium*, Sept. 1985, pp. 172-179.

[Cristian 86] Cristian, F., Aghili, H., Strong, R., "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Tech. Report RJ4540(48668), IBM Research Lab.*, Dec. 1986.

[Crowcroft 88] Crowcroft, J., Paliwoda, K., "A Multicast Transport Protocol," *ACM SIGCOMM'88 Symposium*, Aug. 1988, pp. 247-256.

[Fidge 88] Fidge, C.J., "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," *Proc. of the 11th Australian Computer Science Conference*, Brisbane, Feb. 1988, pp. 56-66.

[Kaashoek 89] Kaashoek, M.F., Tanenbaum, A.S., "An Efficient Reliable Broadcast Protocol," *ACM Symposium on Operating Systems*, Vol. 3, No. 4, Oct. 1989, pp. 5-19.

[Lampor 78] Lamport, L., "Time, Clock and Ordering or Events," *Communications of ACM*, Vol. 21, No. 7, Jul. 1978, pp. 558-565.

[LeBlanc 85] LeBlanc, T.J., Cook, R.P., "High-Level Broadcast Communication for Local Area Networks," *IEEE Software*, Vol. 2, No. 3, May. 1985, pp. 40-48.

[Leffler 85] Leffler, S., Fabry R., Joy, W., "A 4.2BSD Interprocess Communication Primer," *4.2BSD VAX Distribution*,

[Metcal 76] Metcalfe, R.M., Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of ACM*, Vol. 19, Jul. 1976, pp. 395-404.

[Mockapetris 83] Mockapetris, P.V., "Analysis of Reliable Multicast Algorothms for Local Networks," *Proc. of the 8th Data Communication Symposium*, Oct. 1983, pp. 150-157.

[Navaratnam 88] Navaratnam, S., Chanson, S., Neufeld, G., "Reliable Group Communication in Distributed Systems," *Proc. of IEEE the 8th International Conference of Distributed Computing Systems*, Jun. 1988, pp. 439-446.

[Postel 80] Postel, J., "User Datagram Protocol," *USC Inform. Sci. Inst., Rep. RFC 768*, Aug. 1980.

[RFC793 81] "Transmission Control Protocol," *USC Inform. Sci. Inst., Rep. RFC 793*, Sept. 1981.

[RFC791 81] "Internet Protocol – DARPA Internet Program Protocol Specification," *USC Inform. Sci. Inst., Rep. RFC 791*, Sept. 1981.

[Skeen 83] Skeen, D., "Determining the Last Process to Fail," *ACM Transactions on Computer Systems*, Vol 3, No. 1. Feb. 1985,

[Shoja 88] Shoja, G.C., "A Distributed Facility for Load Sharing and Parallel Processing Among Workstations," *DCS-95-IR Computer Science Dept. University of Victoria*, Sep. 1988.

[Side 90] Side, R., "DPD: A Distributed Program Debugger for the REM Environment," Master Thesis, *Computer Science Dept. University of Victoria*, 1990.

[Vuong 89] Vuong, S.T., See, H.L., "Multicast Communication in the Distributed System T-Shoshin," *IEEE Pacific Rim Conference on Communications and Signal Processing*, Jun. 1989, pp. 261-264.

[Wall 80] Wall, D.W., "Mechanisms for Broadcasts and Selective Broadcasts," Ph.D Thesis, *Computer Science, Stanford University*, June, 1980.

[Wong 85] Wong, J.W., Gospal, G., "Reliable Broadcast on Local-Area Networks," *IEEE International Communications Conference*, Vol. 3, Jun. 1985, pp. 43.4.1-43.4.5.

Appendix A

Fidge's Logical Clock

In this appendix we give a brief description of Fidge's partially-ordered clocks [Fidge 88] for asynchronous communication.

A logical clock is maintained in each of n processes in a process cluster. The clock is represented as a vector [c_1, c_2, ..., c_n] with an integer clock value for each process(c_i contains the clock value of process i). During execution, each atomic event is timestamped with the current value of the logical clock after the clock is updated.

Let e_p represent an event e occurring in process p and T_{e_p} represent the timestamp vector of the event e_p. The i^{th} element of T_{e_p} is denoted by $T_{e_p}[i]$.

For asynchronous communication, the logical clock is maintained as follows:

C_1 : Initially all clock values are set to the smallest value.

C_2 : The logical clock value is incremented at least once before each primitive event in a process.

C_3 : The current value of the entire logical clock vector is delivered to the receiver for every outgoing message.

C_4 : Upon receiving a message, the receiver sets the value of each entry in its local timestamp vector to the maximum of the two corresponding values in the local vector and in the remote vector received. The element corresponding to the sender is a special case; it is set to one greater than the value received, but only if the local value is not greater than that received.

C_5 : Values in the timestamp vectors are never decremented.

Timestamps attached to the events are compared as follows:

$$e_p \rightarrow f_q \textbf{ iff } T_{e_p}[p] < T_{f_q}[p],$$

This means event e_p happened before event f_q if and only if process q received a direct or indirect message from p and that message was sent after e_p had occurred. If e_p and f_q are in the same process ($i.e., p = q$), the local elements of their timestamps represent their occurrences in the process.

Figure A.1 is shows how events are timestamped at runtime.

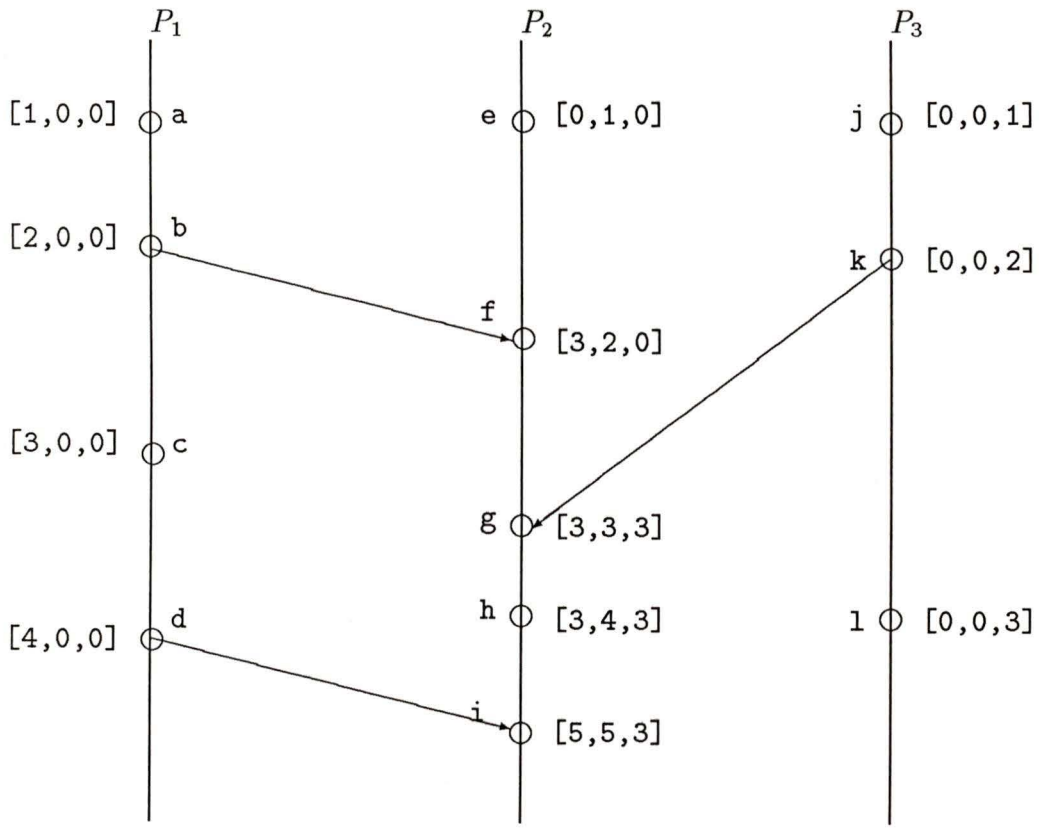


Figure A.1: **An example of Fidge event timestamps**

Appendix B

UNIX sockets

Sockets are the endpoints of a communication path and have three attributes: socket domain, socket type, and protocol family.

(a) Socket domain

A socket in use usually has an *address* bound to it. The nature of the address depends on the *communication domain* of the socket. Processes communicating in the same domain use the same *address format.* A single socket can communicate in only one domain.

There are two domains in 4.2BSD: Unix domain (AF_UNIX) and the Internet domain (AF_INET).

- Unix domain: When both ends of a socket connection are in the same computer, we use Unix domain in which the address format is ordinary file system pathnames.
- Inet domain: Internet domain is used by processes which communicate using the DARPA standard communication protocols. The internet addresses are used as socket address format.

(b) Socket type

Sockets are typed according to the communication properties visible to a user. There are several kinds of socket type. We only concentrate on the two of them used in our environment, namely the stream socket and the datagram socket.

- Stream socket

  Stream sockets provides reliable, full-duplex, sequenced data streams. No data is lost or duplicated in delivery, and there are no record boundaries.

- Datagram socket

  Datagram sockets supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated but the record boundaries in data are preserved.

(c) Protocol family

A user can select a protocol to associate with a socket while creating it. In the Internet domain, stream sockets are supported by TCP (Internet Transmission Control Protocol) and datagram sockets are supported by UDP (Internet User Datagram Protocol). TCP ensures that messages arrive at the receiving process in the order they are sent without loss or duplication. UDP does not guarantee the arrival of the message to the intended receiving process.

In Berkeley UNIX, a set of system calls of socket operation are provided to the users. Table B.1 is a list of these calls.

names	function descriptions
socket	Create a socket of a given type, domain and protocol
bind	Associate an ASCII name to a previously created socket
listen	Create a queue to store incoming connection requests
accept	Remove a connection request from the queue or wait for one
connect	Initiate a connection with a remote socket
select	Check a set of sockets to see if any can be read or writen
send	Send a message through a stream socket
recv	Receive a message on a stream socket
sendto	Send a message through a datagram socket
recvfrom	Receive a message on a datagram socket
close	Terminate a socket
shutdown	Terminate the connection on a socket

Table B.1: **The principle socket operations in Berkeley UNIX**

Appendix C

Glossary

This appendix gives a description of the list of abbreviations used in this thesis.

abbreviations	**descriptions**
IP	Internet Protocol
IPC	Inter-process Communication
LAN	Local Area Network
REM	Remote Execution Manager
TCP	Transmission Control Protocol
UDP	User Datagram protocol
WAN	Wide Area Network

Table C.1: **Abbreviations of general terminology**

Appendix D

Benchmark Code

D.1 CPU intensive task

D.1.1 Unicast

[1]

Parent process

```
/*
 * Test cpu intensive program, parent program
 *
 * cpu.c
 */

#include <stdio.h>
#include <sys/time.h>
#include "globals.h"
#include "timer.h"
#include "dist.h"
```

[1]This code was originally written by G. Clark. It was modified using our multicasting primitives.

```
#define round(x,y) (int) ((double)x /(double)y + 0.5)

#define MAX_KIDS 100
#define ITT_COUNT 1000

t_time the_time;

int num_kids;
int loops, i, j, sum, kid_loops[MAX_KIDS];

int sock_val, sm_pid;

double kid_array[MAX_KIDS];
double x1=1.0,
y1=1.0,
z1=1.0;

char my_rem[]= DEFAULT_SUITE;


/*************************************************************/
main(argc, argv)
int      argc;
char    *argv[];
{
char *my_name=argv[0],
        *char_loops=argv[1],
        *char_kids=argv[2];

    if (argc < 3) {
        printf("Error: Usage is %s number_of_loops number_of_kids %d\n",
my_name, argc );
        exit(-1);
    }


    loops = atoi(char_loops);
    num_kids = atoi(char_kids);

/* uniprocessor */
if (num_kids == 0) { /* uniprocessor */
```

```
Start(the_time);
for (i=0; i<loops; ++i)
for (j=0; j < ITT_COUNT; ++j)
x1=y1/z1;
Stop(the_time);

    putchar('\n');
    printf("\n%d %d  ", loops, num_kids);
    Display(the_time);
    printf("\t");
    Display_user(the_time);
    printf("\t");
    Display_sys(the_time);
    printf("\n");

exit(0);
}

/* multiprocessor */

     initialize_dist(my_rem);            /* contact process manager      */

for (i=0; i<num_kids; ++i) {        /* start receiving child        */
         kid_array[i]=create_dist("cpu_child");
}

sum=j=0;    /* will be used to add up kids   */

i = round(loops, num_kids);

for (j=0; j<num_kids-1; loops-=i, ++j) /*distribute number of loops */
    kid_loops[j] = i;

kid_loops[j] = loops; /* last child does leftover work*/

Start(the_time);

for (i=0; i<num_kids; ++i) /* send the number of loops  */
         send_dist(kid_array[i], &kid_loops[i], sizeof(int));

for (i=0; i<num_kids; ++i) {    /* get back the answers */
         us_recv_dist(kid_array[i], (char *) &j, sizeof( j ));
```

```
sum += j;
}
Stop(the_time);

    putchar('\n');
    printf("\n%d %d  ", atoi(char_loops), num_kids);
    printf("\n%d %d  ", sum, num_kids);
    Display(the_time);
    printf("\t");
    Display_user(the_time);
    printf("\t");
    Display_sys(the_time);
    printf("\n");

    terminate_dist();          /* clean up    */
}
```

Child process

```
/*
 * Test cpu intensive program, child program
 *
 * cpu_child.c
 */

#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include "globals.h"
#include "dist.h"

#define ITT_COUNT       1000

int loops, i, j, k=1;
double x1=1.0,
y1=1.0,
z1=1.0,
parent;

char my_rem[]= DEFAULT_SUITE;
```

```
/*********************************************************/
main()
{
    initialize_dist(my_rem); /* contact process manager  */
parent = getdistppid(); /* get parents pid */

us_recv_dist(parent, &loops, sizeof( loops )); /* get # of loops */

printf("loops %d\n", loops);

for (i=0; i<loops; ++i) /* do specified loops 1 */
    for (j=0; j < ITT_COUNT; ++j)
        x1=y1/z1;

send_dist(parent, &k, sizeof(int)); /* send sync message */


    terminate_dist();                    /* clean up            */
}
```

D.1.2 Weak reliable multicast

Parent process

```
/*
 * Test cpu intensive program, parent program
 *
 * mcpu.c (weak multicast version)
 */

#include <stdio.h>
#include <sys/time.h>
#include "globals.h"
#include "timer.h"
#include "dist.h"
#include "mcast.h"
```

```
#define round(x,y) (int) ((double)x /(double)y + 0.5)

#define MAX_KIDS 100
#define ITT_COUNT 1000

t_time the_time;

int num_kids, gsize, mc_ret_val;
int loops, i, j, sum, kid_loops[MAX_KIDS];

double kid_array[MAX_KIDS];
double ppid;

double x1=1.0,
y1=1.0,
z1=1.0;

char my_rem[]= DEFAULT_SUITE;

/*************************************************************/
main(argc, argv)
int      argc;
char    *argv[];
{
char *my_name=argv[0],
        *char_loops=argv[1],
        *char_kids=argv[2];

    if (argc < 3) {
        printf("Error: Usage is %s number_of_loops number_of_kids %d\n",
my_name, argc );
        exit(-1);
    }


    loops = atoi(char_loops);
    num_kids = atoi(char_kids);
    gsize = num_kids + 1;

/* uniprocessor */
if (num_kids == 0) { /* uniprocessor */
Start(the_time);
```

```
for (i=0; i<loops; ++i)
for (j=0; j < ITT_COUNT; ++j)
x1=y1/z1;
Stop(the_time);

    putchar('\n');
    printf("\n%d %d  ", loops, num_kids);
    Display(the_time);
    printf("\t");
    Display_user(the_time);
    printf("\t");
    Display_sys(the_time);
    printf("\n");

exit(0);
}

/* multiprocessor */

     initialize_dist(my_rem);            /* contact process manager      */

ppid = getdistppid();

initialize_s_mcast(gsize);       /* inform REM the GSIZE*/

for (i=0; i<num_kids; ++i) {        /* start receiving child        */
         kid_array[i]=mcreate_unrel("mcpu_child");
}

sum=j=0;    /* will be used to add up kids   */

i = round(loops, num_kids);

for (j=0; j<num_kids-1; loops-=i, ++j) /*distribute number of loops */
    kid_loops[j] = i;

Start(the_time);

/* send the number of loops to all the kids */
     mc_ret_val = msend(ppid, &kid_loops[0], sizeof(int));

for (i=0; i<num_kids; ++i) {    /* get back the answers */
```

```
        us_recv_dist(kid_array[i], (char *) &j, sizeof( j ));
sum += j;
}
Stop(the_time);

    putchar('\n');
    printf("\n%d %d  ", atoi(char_loops), num_kids);
    printf("\n%d %d  ", sum, num_kids);
    Display(the_time);
    printf("\t");
    Display_user(the_time);
    printf("\t");
    Display_sys(the_time);
    printf("\n");

    mremove();          /* terminate */
} /* mcpu.c */
```

Child process

```
/*
 * Test cpu intensive program, child program
 *
 * mcpu_child.c (multicast version)
 */

#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include "globals.h"
#include "dist.h"
#include "mcast.h"

#define ITT_COUNT       1000

int loops, i, j, k=1;

int mesg_len, mc_ret_val;
```

```
double x1=1.0,
y1=1.0,
z1=1.0,
parent;

char my_rem[]= DEFAULT_SUITE;

/*************************************************************/
main()
{
    initialize_dist(my_rem); /* contact process manager  */
parent = getdistppid(); /* get parents pid */

initialize_r_mcast(0);      /* unreliable comm */

mesg_len = mrecv(parent, &loops, sizeof( loops )); /* get # of loops */

printf("loops %d\n", loops);

for (i=0; i<loops; ++i) /* do specified loops 1 */
    for (j=0; j < ITT_COUNT; ++j)
        x1=y1/z1;

send_dist(parent, &k, sizeof(int)); /* send sync message */


    mleave();        /* terminate */
} /* mcpu_child */
```

D.1.3 Strong reliable multicast

(a) Parent process

```
    /*
     * Test cpu intensive program, parent program
     *
     * rcpu.c (multicast version with reliable protocol)
```

```
 */

#include <stdio.h>
#include <sys/time.h>
#include "globals.h"
#include "timer.h"
#include "dist.h"
#include "mcast.h"

#define round(x,y) (int) ((double)x /(double)y + 0.5)

#define MAX_KIDS 100
#define ITT_COUNT 1000

t_time the_time;

int num_kids, gsize, mc_ret_val;
int loops, i, j, r, sum, kid_loops[MAX_KIDS];

double kid_array[MAX_KIDS];
double ppid;

double x1=1.0,
y1=1.0,
z1=1.0;

char my_rem[]= DEFAULT_SUITE;

/*************************************************************/
main(argc, argv)
int      argc;
char    *argv[];
{
char *my_name=argv[0],
        *char_loops=argv[1],
        *char_kids=argv[2];

FILE *fopen(), *fp_rcpu;

    if (argc < 3) {
        printf("Error: Usage is %s number_of_loops number_of_kids %d\n",
my_name, argc );
```

```
        exit(-1);
    }

    loops = atoi(char_loops);
    num_kids = atoi(char_kids);
    gsize = num_kids + 1;

/* uniprocessor */
if (num_kids == 0) { /* uniprocessor */
Start(the_time);
for (i=0; i<loops; ++i)
for (j=0; j < ITT_COUNT; ++j)
x1=y1/z1;
Stop(the_time);

     putchar('\n');
     printf("\n%d %d  ", loops, num_kids);
     Display(the_time);
     printf("\t");
     Display_user(the_time);
     printf("\t");
     Display_sys(the_time);
     printf("\n");

exit(0);
}

/* multiprocessor */

     initialize_dist(my_dpm);            /* contact process manager      */

ppid = getdistppid();

initialize_s_mcast(gsize);       /* inform REM the GSIZE*/

for (i=0; i<num_kids; ++i) {        /* start receiving child        */
         kid_array[i]=mcreate_relia("rcpu_child");
}

sum=j=0;    /* will be used to add up kids   */

i = round(loops, num_kids);
```

```
for (j=0; j<num_kids-1; loops-=i, ++j) *distribute number of loops *
    kid_loops[j] = i;

Start(the_time);

/* send the number of loops to all the kids */

     mc_ret_val = msend(ppid, &kid_loops[0], sizeof(int));

for (i=0; i<num_kids; ++i) {    /* get back the answers */
         us_recv_dist(kid_array[i], (char *) &j, sizeof( j ));
sum += j;
}
Stop(the_time);

     Display(the_time);
     printf("\t");
     Display_user(the_time);
     printf("\t");
     Display_sys(the_time);
     printf("\n");

        mremove();           /* terminate */

} /* rcpu.c */
```

(b) Child process

```
/*
 * Test cpu intensive program, child program
 *
 * rcpu_child.c (multicast version with reliable protocol)
 */

#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include "globals.h"
#include "dist.h"
```

```
#include "mcast.h"

#define ITT_COUNT       1000

int loop, i, j, k=1;

int mesg_len, mc_ret_val;

double x1=1.0,
y1=1.0,
z1=1.0,
parent;

char my_rem[]= DEFAULT_SUITE;

/*************************************************************/
main()
{
    initialize_dist(my_dpm); /* contact process manager  */
parent = getdistppid(); /* get parents pid */

initialize_r_mcast(1);      /* reliable comm */

mesg_len = mrecv(parent, &loops, sizeof( loops ));

for (i=0; i<loops; ++i) /* do specified loops  */
    for (j=0; j < ITT_COUNT; ++j)
        x1=y1/z1;

send_dist(parent, &k, sizeof(int)); /* send sync message */

    } /* loop */

    mleave();      /* terminate */

} /* rcpu_child.c */
```

D.2 Message passing

D.2.1 Unicast

Parent process

```
%test_send.c
/*
 * Test sending of messages, parent program
 *
 * ucast_send.c
 */

#include <stdio.h>
#include <sys/time.h>
#include "globals.h"
#include "dist.h"
#include "timer.h"

#define MAX_KIDS        14
int pkts, size, kids, i, j;
char foo[4000];
char info_buf[80];

char my_rem[]= DEFAULT_SUITE;

double kid_array[MAX_KIDS];

t_time the_time;

/*************************************************************/
main(argc, argv)
int             argc;
char    *argv[];
{
register char   *my_name=argv[0],
        *char_pkts=argv[1],
        *char_size=argv[2],
        *char_kids=argv[3];
```

```
char test_line[128];

    if (argc < 3) {
        printf("Error: Usage is %s number_of_pkts message_size num_of_kids\n", n
        exit(-1);
    }

    pkts = atoi(char_pkts);
    size = atoi(char_size);
    kids = atoi(char_kids);

    initialize_dist(my_rem);    /* contact process manager */
    for (i=0; i<kids; i++) {
            kid_array[i]=create_dist("ucast_recv");
         printf("### kid=create_dist = %d ###n", kid_array[i]);
    }

    strcpy (test_line, "** Hello from the Parent process. **");

    info_buf[0] = '\0';
    sprintf(info_buf, "%d %d \0", pkts, size);
    printf("\n**outbuf = !%s!\n", info_buf);

    Start(the_time);

    for(i=0; i<kids; i++) {
     send_dist(kid_array[i], test_line, strlen(test_line)+1);

     send_dist(kid_array[i], info_buf, strlen(info_buf));

     for (j=pkts; j>0; --j) {       /*do all of the sends */
         send_dist(kid_array[i], foo, size);
     }
    }

    Stop(the_time);

    terminate_dist();                       /* clean up                    */

    printf("\tpkts\tsize\tkids\n");
    printf("\t%d\t%d\t%d\t", pkts, size, kids);
```

```
    printf("\n");
    putchar('\n');
    Display(the_time);
    printf("\t");
    Display_user(the_time);
    printf("\t");
    Display_sys(the_time);
    printf("\n");

} /* ucast_send.c */
```

Child process

```
%test_send_child.c
/*
 * Test sending of messages, child program
 *
 * ucast_recv.c
 */


#include <stdio.h>
#include "globals.h"
#include "dist.h"

char my_rem[]= DEFAULT_SUITE;

char foo[4000];
char input_buf[80];

double ppid;
int i, pkts, size;

main()
{
char test_line [128];

    printf ("** child calling initialize_dist. **\n");
```

```
    initialize_dist(my_rem);  /* contact process manager  */
    printf ("** successful init. **\n");

    ppid=getdistppid();          /* get parent pid !          */
    printf("### ppid=getdistppid = %d ###n", ppid);
    printf ("\n*** CHILD about to receive.\n");
    us_recv_dist(ppid, test_line, 128);
    printf ("%s\n", test_line);

    us_recv_dist(ppid, input_buf, 80);
    sscanf(input_buf, "%d %d %d", &pkts, &size);
    printf("\n**Incoming buffer: !%s!\n", input_buf);

    printf ("\n*** child reading %d packets of size %d.\n", pkts, size);

    for(i=pkts; i>0; --i) { /* receive all pkts */
printf ("\n*** child recieving %d.\n", i);
us_recv_dist(ppid, foo, 4000);
    }

    terminate_dist();
} /* ucast_recv.c */
```

D.2.2 Multicast with weak reliability

Parent process

```
/*
 * Test multicast sending of messages, parent program
 *
 * mcast_send.c
 */

#include <stdio.h>
#include <sys/time.h>
#include "globals.h"
#include "dist.h"
#include "timer.h"
```

```
#include "mcast.h"

char my_rem[]= DEFAULT_SUITE;
#define MAX_KIDS        14

int pkts, size, kids, i, j;
int gsize, mc_ret_val;
double ppid;

char foo[1000];
char info_buf[80];

double kid_array[MAX_KIDS];

t_time the_time;

/*************************************************************/
main(argc, argv)
int             argc;
char    *argv[];
{
register char   *my_name=argv[0],
        *char_pkts=argv[1],
        *char_size=argv[2],
        *char_kids=argv[3];
char test_line[128];

    if (argc < 3) {
        printf("Error: Usage is %s number_of_pkts message_size num_of_kids\n", m
        exit(-1);
    }

    pkts = atoi(char_pkts);
    size = atoi(char_size);
    kids = atoi(char_kids);

    gsize = kids + 1;

    initialize_dist(my_rem);    /* contact process manager */

    ppid = getdistppid();
```

```
        initialize_s_mcast(gsize);  /* inform REM the GSIZE */

        for (i=0; i<kids; i++) {
                kid_array[i]=mcreate_unrel("mrecv");
        }

        strcpy (test_line, "** Hello from the Parent process. **");

        info_buf[0] = '\0';
        sprintf(info_buf, "%d %d \0", pkts, size);
        printf("\n**outbuf = !%s!\n", info_buf);

        Start(the_time);

        mc_ret_val = msend(ppid, test_line, strlen(test_line)+1);

        mc_ret_val = msend(ppid, info_buf, strlen(info_buf));

        for (j=pkts; j>0; --j) {        /*do all of the sends */
            mc_ret_val = msend(ppid, foo, size);
        }

        Stop(the_time);

        mremove();              /* terminate */

        printf("\tpkts\tsize\tkids\n");
        printf("\t%d\t%d\t%d\t", pkts, size, kids);

        printf("\n");
        putchar('\n');
        Display(the_time);
        printf("\t");
        Display_user(the_time);
        printf("\t");
        Display_sys(the_time);
        printf("\n");

    } /* mcast_send.c */
```

Child process

```
/*
 * Test multicast sending of messages, child program
 *
 * mcast_recv.c
 */


#include <stdio.h>
#include "globals.h"
#include "dist.h"
#include "mcast.h"

char my_rem[]= DEFAULT_SUITE;

char foo[1000];
char input_buf[80];

double ppid;
int i, pkts, size;
int mc_ret_val;

main()
{
char test_line [128];

    printf ("** child calling initialize_dist. **\n");

    initialize_dist(my_rem);  /* contact process manager  */
    printf ("** successful init. **\n");

    ppid=getdistppid();          /* get parent pid !          */
    printf("### ppid=getdistppid = %d ###n", ppid);

    initialize_r_mcast(0);       /* unreliable comm */

    printf ("\n*** CHILD about to receive.\n");
    mc_ret_val = mrecv(ppid, test_line, 128);
    printf ("%s\n", test_line);
```

```
    mc_ret_val = mrecv(ppid, input_buf, 80);
    sscanf(input_buf, "%d %d %d", &pkts, &size);
    printf("\n**Incoming buffer: !%s!\n", input_buf);

    printf ("\n*** child reading %d packets of size %d.\n", pkts, size);

    for(i=pkts; i>0; --i) { /* receive all pkts */
printf ("\n*** child recieving %d.\n", i);
mc_ret_val = mrecv(ppid, foo, 1000);
    }

    mleave();
} /* mcast_recv.c */
```

VITA

Surname: **Liu** Given Names: **Frances Fang**
Place of Birth: **Beijing, China** Date of Birth: **April 28, 1961**

Educational Institutions Attended:

The Second Branch of Beijing University	1979 to 1983
University of Victoria	1988 to 1990

Degrees Awarded:

B.Sc.	1983	The Second Branch of Beijing University

Honors and Awards:

University of Victoria Fellowship, 1987/89

Partial Copyright License

Title of Thesis:

Reliable Multicast Communication For Parallel Computation in Distributed Environments

Author: [redacted]
Frances F. Liu
August 8, 1990