UNIVERSITY OF NEWCASTLE UPON TYNE

DEPARTMENT OF COMPUTING SCIENCE

FAULT-TOLERANT GROUP COMMUNICATION PROTOCOLS FOR ASYNCHRONOUS SYSTEMS

Ph.D THESIS

BY

Raimundo José de Araújo Macêdo

NEWCASTLE UPON TYNE

AUGUST, 1994

I dedicate this thesis to my children

Lucas and Laís

and to my parents

Joselita and Raimundo

Acknowledgements

I would like to express my sincere gratitude to several people who have contributed in various ways to the completion of this thesis.

First and foremost, I thank my supervisor, Professor Santosh K. Shrivastava. His guidance and technical contributions throughout this work have been invaluable. Santosh's constant enthusiasm and encouragement have been essential for the completion of this thesis.

I am indebted to Dr. Paul Ezhichelvan. The countless discussions I have had with Paul have helped me to clarify many questions on Group Communication Protocols. I also declare that the material on group membership presented in this thesis has been carried out through joint work with Paul.

I shall never forget the support and friendship of many people who along these last few years have helped me to overcome many difficult moments of my personal life. Although taking the risk of omitting important names, I shall mention Claudete Alves, Marta Macêdo, Aldineia Ferreira, German Medina, Marisol Kucharek, Marcos Euzebio, Sandrine Dalban, Carlos Lopes, Kristina Baldus, Robert Burnett, Tatiana Simas, and Helder Pires.

I would like also to thank several of my colleagues and staff members of the Computing Science Department for their prompt help and technical support on many occasions. These include Trevor Kirby, Alcides Calsavara, Fernando Capretz, Rogerio de Lemos, Luis Buzato, Stuart Wheater, and Gillian Dobson. Also, many thanks to Shirley Craig for her patience and efficient help in searching out many relevant references for this thesis.

I am also thankful to the Federal University of Bahia, in Brazil, which provided institutional support for my studies in Newcastle. Financial support for this thesis has been provided by the Brazilian Research Council (CNPq), grant number 200811/89.4.

Abstract

It is widely accepted that group communication (multicast) is a powerful abstraction that can be used whenever a collection of distributed processes cooperate to achieve a common goal such as load-sharing or fault-tolerance. Due to the uncertainties inherent to distributed systems (emerging from communication and/or process failures), group communication protocols have to face situations where, for instance, a sender process fails when a multicast is underway or where messages from different senders arrive in an inconsistent order at different destination processes. Further complications arise if processes belong to multiple groups.

In this thesis, we make use of logical clocks [Lamport78] to develop the concept of Causal Blocks. We show that Causal Blocks provide a concise method for deducing ordering relationships between messages exchanged by processes of a group, resulting in simple methods for dealing with multiple groups. Based on the Causal Blocks representation, we present a protocol for total order message delivery which has constant and low message space overhead (i.e. the protocol related information contained in a multicast message is small). We also present causal order protocols with different trade-offs between message space overhead and speed of message delivery. Furthermore, we show how the Causal Blocks representation can be used to easily deduce and maintain reliability information. Our protocols are fault-tolerant: ordering and liveness are preserved even if group membership changes occur (due to failures such as process crashes or network partitions). The total order protocol, together with a novel flow control mechanism, has been implemented over a set of networked Unix workstations, and experiments carried out to analyse its performance in varied group configurations.

Contents

Illustrations

Chapter 1 - Introduction

Distributed systems are generally characterised by a set of processes residing at a number of computing units (such as workstations or personal computers), possibly spread over a geographical area, where processes communicate to each other only by means of message exchange. The use of such systems have notably expanded over the years, for computing units and communication links have become increasingly more available and cheaper.

Besides making possible the realisation of applications that are inherently distributed (e.g. computer supported collaborative work), distributed systems facilitate two new possibilities for computing systems, when compared with the traditional centralised systems: the possibility of implementing fault-tolerance by replicating processes over distinct computing units (in contrast to just hardware redundancy used in centralised systems) and the exploitation of parallel computations to improve application performance. Unfortunately, there are two difficulties that must be faced by designers of distributed systems which are not present for centralised systems. Firstly, the absence of global state (global clock or shared memory) among distributed processes makes the problem of ordering events in a distributed computation become much harder. Secondly, time uncertainties on process communication delays caused by factors such as communication failures (e.g. message losses) or arbitrary process execution times due to variations on system loads, introduces a new problem: how to detect whether a remote process is operational (i.e. whether or not it has crashed)[1]. In a group communication setting, such an uncertainty can create inconsistencies (processes can get mutually inconsistent views of the group membership). System designers are then faced with the challenge of providing proper abstractions which would hide most of the common complexity of

[1] Note that Distributed Systems can partially fail due to the failures of some of its components (processes or communication links) whereas centralized systems only fail entirely.

distributed programming, facilitating the work of application developers. Group communication with some message ordering and reliability guarantees have been advocated as such an abstraction [Birman91a, Olsen91, Verissímo93] and it is the concern of this thesis.

1.1 Group Communication

Group communication (or multicast) can be used whenever groups of distributed processes cooperate for the execution of a given task such as committing a distributed data base transaction, or to achieve fault-tolerance or better performance (by replication). In group communication, processes usually communicate in a group basis where a message is sent to a group of processes, rather than just to one process, which is the case in point-to-point communication. With a group is usually associated a name to which application processes will refer, making transparent the location of the distributed processes forming the group. Due to the uncertainties inherent to distributed systems (emerging from communication or process failures), group communication protocols have to face situations where, for instance, a sender process fails when a multicast is underway or where messages arrive in an inconsistent order at different destination processes. On the other hand, distributed applications usually require that processes forming a group "see" events such as processes failures and message delivery in a mutually consistent way. For instance, active replication requires that messages are delivered in the same order at all the replicas and that failures are handled in a mutual consistent manner among operational replicas.

1.1.1 Process Crashes and Membership Reconfiguration

Process crashes should ideally be handled by a fault tolerant multicast protocol in the following manner: when a process does crash, all functioning processes must promptly observe that crash event and agree on the order of that event relative to other events in the system. As hinted earlier, in systems subjected to arbitrary communication and processing delays (such a system is referred to as an asynchronous system) this is impossible to achieve: when processes are prone to

failures, it is impossible to guarantee that all non-faulty processes will reach agreement in finite time [Fischer85]. This impossibility stems from a process' inability to distinguish slow processes from crashed ones. Asynchronous protocols therefore need to circumvent this impossibility result by permitting processes to *suspect* process crashes [Chandra91] and to reach agreement only among those processes which they do not suspect to have crashed [Ricciardi91]. A process detected to have crashed will then be removed from the group membership of the operational members involved in the agreement. Hence, a membership reconfiguration service must work in co-operation with the group communication protocol in order to provide the members of a group with a mutually consistent view of the order in which membership changes occur. Moreover, processes forming a new membership should be delivered the same set of messages, even when a process removed from the membership crashed during a multicast and before that multicast message had reached all destination processes.

1.1.2 Message Ordering

Some distributed applications require that messages multicast are delivered in such a way that some specific order is not violated. In asynchronous systems, messages may arrive in arbitrary order at destinations. For example, consider that the messages m and m' are sent by two different processes to a group g. A given process p belonging to g could receive m before m' whereas another process q also belonging to g could received m' before m. Thus, delivery order of m and m' in p and q would violate the identical order principle required, for instance, by active replication applications. So, group communication protocols have to provide users with some message order guarantees, making transparent to the application processes the fact that messages may be delivered in an arbitrary order by the low level network protocol.

1.1.3 Message Delivery in Overlapping Process Groups

In some applications, processes are required to simultaneously participate in multiple groups [Birman91c, Garcia-Molina91]. That is, process groups may overlap. For example, take a computer based conferencing application where users may simultaneously participate in different conversations (or groups). Suppose a given multi-group user generates a message m' in a group B as a consequence of a message m delivered to the same user in group A. Other multi-group users participating simultaneously in groups A and B, would then require that m' be delivered only after m has been delivered (otherwise, m' may make no sense). Since message m potentially caused message m', we say that they are causally related [Lamport78]. For correct delivery of messages m and m', we require a protocol which delivers messages respecting their causal origin or in causal order. If besides respecting the causal order, messages are required to be delivered in identical order, we say that messages are required to be delivered in total order. The precise notion of causality as well as other order requirements for overlapping groups are examined in more detail in the next chapter as well as in the subsequent chapters where our protocols are presented.

1.1.4 Existing Group Communication Protocols

In the last few years, several group communication protocols have been proposed for use in distinct system settings, providing users with a variety of services [Chang84, Birman91b, Peterson89, Melliar-Smith90, Amir92a]. This thesis builds on existing works and is concerned with the development of portable (i.e. not designed for a specific network topology or system setting) fault-tolerant group communication protocols. Particularly we aim for a protocol adequate to a variety of network topologies and application requirements such as group overlapping. Thus, we avoid specific assumptions such as known and bounded message transmission latency [Lamport82, Strong83, Fischer83, Babaoglu88], broadcast support on the underlying communication network [Melliar-Smith90, Amir92a, Rodrigues91], or the absence of network partitions [Birman91b, Peterson89]. In addition, we aim for protocols that

preserve some required message order delivery even when groups overlap. Although group overlapping has been addressed in previous work [Birman91b, Garcia-Molina91, Mostefaoui93], these protocols have not addressed causality preserving total order message delivery for overlapping groups.

1.2 Contributions of the Thesis

In this thesis, we make use of logical clocks [Lamport78] to develop the concept of Causal Blocks. We show that Causal Blocks provide a concise method for deducing ordering relationships between messages exchanged by processes of a group, resulting in simple methods to deal with overlapping groups. Based on the Causal Blocks representation, we present a protocol for total order message delivery which has constant and low message space overhead (i.e. the protocol related information contained in a multicast message is small). We also present causal order protocols with different trade-offs between message space overhead and speed of message delivery. Furthermore, we show how the Causal Blocks representation can be used to easily deduce and maintain reliability information.

The protocols we present do not use extra control messages in order to enforce correct message delivery. Some of the protocols, however, require that processes are lively in sending messages (i.e. now and then a new message is transmitted), and to guarantee this liveness we introduce the *time-silence* mechanism.

We develop and show the implementation of a distributed failure suspector based on the Causal Blocks representation and local time-outs. Based on this failure suspector, we have developed a fault-tolerant total order protocol for overlapping process groups which works correctly despite process crashes and network partitions. To our best knowledge there is no other existing protocol fulfilling these requirements all together. We also present a novel flow control mechanism to be used in multicast protocols and formally prove its correctness. All protocols and services such as membership, the flow control, *time-silence*, and the suspector have been developed based on the Causal Blocks representation and therefore they work in a integrated

manner, without relying on any external service. This makes our work distinctive from existing works in the area.

We have implemented the total order protocol (Newtop) and carried out experiments to analyse the effects of the *time-silence* mechanism (transmission of null messages) and the degree of processes activity (rate of message transmission) on the protocol performance.

1.3 Thesis outline

This thesis is structured in 9 chapters. Basic background material is presented in chapter 2. Chapter 3 introduces the Causal Blocks model and it is a pre-requisite for all remaining chapters. Chapters 4 and 5 are independent and can be read in any order. Chapter 6 and 7 can also be read in any order but their contents depend on chapter 4. Chapter 8, if desirable, can be read straight after chapter 3. Below is a summary of the chapter contents, from chapter 2 to chapter 9.

Chapter 2 discusses issues related with Group Communication. Section 2.1 comments on the effects of synchronous and asynchronous modelling on performance and failure detection. Section 2.2 presents the system model. The following three sections discusses issues related with group communication protocols: group overlapping, message order delivery, and fault-tolerance, respectively. Section 2.6 comments on previous related work and section 2.7 concludes the chapter.

Chapter 3 presents the basic principles and terminology of the Causal Blocks model which has been developed to represent message ordering and reliability information in a concise manner. The first section of the chapter states the system model assumed. The next three sections show how Causal Blocks are constructed, demonstrating the main properties of the model. Section 3.5 shows a compact representation for a Block Matrix (i.e. a set of Causal Blocks). Message ordering for delivery is examined in section 3.6. Sections 3.7 shows how missing (sent but not received) messages are represented in a Block Matrix and section 3.8 concludes the chapter.

In chapter 4 we present a total order protocol for overlapping process groups and prove its correctness. After showing how to organise Causal Blocks for total order delivery in section 4.2, we present in section 4.3 a mechanism called *time-silence* that guarantees the liveness of the protocol. Section 4.4 discusses total order using Causal Blocks in a group overlapping scenario. Section 4.5 gives an algorithmic description and proves correctness of the total order protocol. The chapter is concluded in section 4.6.

In chapter 5 we discuss causal order delivery in overlapping process groups and present protocols based on Causal Blocks which yield different trade-offs between message delivery delay and message space overhead (the amount of ordering information added to user messages). In section 5.1 we show how Causal Block numbers can be used to precisely represent causal relationship between transmitted messages. Sections 5.2 discusses causal order delivery in overlapping process groups. After presenting the system model and failure assumptions in section 5.3, we describe the Slow, Fast, and Relative causal order protocols, in sections 5.4, 5.5, and 5.6, respectively. Section 5.7 concludes the chapter.

Chapter 6 discusses how to introduce fault-tolerance to Causal Blocks based protocols. In particular, we show how to extend the total order protocol presented in chapter 4 into a fault-tolerant total order protocol, by developing a failure suspector and a membership algorithm using the Causal Blocks representation. Section 6.1 discusses the requirement of group partitioning for multicast protocols. Section 6.2 presents the fault-tolerant properties of the total order protocol we develop. Section 6.3 describe the fault-tolerant mechanism developed. Section 6.4 discusses related work. The chapter is concluded in section 6.5.

Chapter 7 describes the implementation of the total order protocol (the protocol Newtop) over a Unix network and shows performance figures collected. Section 7.1 presents how the transport multicast layer has been implemented using Unix Sockets. Section 7.2 describes the implementation of the Newtop protocol. Section 7.3

discusses the experiments realised, shows and comments performance figures. Conclusions are drawn in section 7.4.

Chapter 8 presents a novel flow control mechanism for multicasts and presents its implementation as part of the protocol Newtop. Section 8.1 presents the flow control scheme and proves its correctness. Section 8.2 presents and discusses experimental results. Section 8.3 comments on related work and section 8.4 concludes the chapter.

Conclusions of the thesis are drawn in chapter 9 where future work is also discussed.

Chapter 2 - Group Communication Protocols and Related Problems

In this chapter we discuss the relevant issues related to the design of group communication protocols. The difficulty in developing a group communication protocol will depend on the degree of reliability and ordering guarantees required. Additionally, the processes forming a group should behave as if they were a single entity, so that application processes can deal with a process group without worrying about both its internal structure and the management of group members. We start in the next section by discussing the implications of synchronous and asynchronous communication environment on group communication. We then present the system model adopted in this thesis, discuss issues of ordering and fault-tolerance concerning multicast protocols, and then finally we comment on existing related work.

2.1 Synchrony and Group Communication

Distributed systems can be broadly classified into two categories : synchronous and asynchronous. Synchronous systems assume that message processing and communication delays are known and bounded. Therefore, it is possible to determine timeout durations that can be used as an accurate indication of failures. It is also possible to keep local process's clocks synchronised within a known bound ε and synchronous protocols often rely on this assumption. Consider that δ, $\delta > 0$, is the bounded time interval within which a transmitted message is received at a destination process. Synchronised clocks imply that for any two processes *p* and *q*, with local clocks C_p and C_q, respectively, and for a given moment in real-time t, $| C_p(t) - C_q(t) | \leq \varepsilon$. So, if there are no failures, a transmitted message timestamped with the sender process' clock reading, say T, will arrive at the destination process no later than $T + \delta + \varepsilon$ (according to the local clock of the destination process). There are also protocols for synchronous systems which do not rely on the assumption of explicitly

synchronised clocks but still guarantee that operations are terminated within a bounded time interval [Ezhilchelvan93, Rodrigues91].

Multicast protocols for synchronous systems [Lamport82, Strong83, Fischer83, Babaoglu88, Cristian90][1] usually work in a scheme of rounds where a round has a time duration, say Δ, (according to process local clocks). Messages are sent in the beginning of a round, and Δ must be "large enough" so that, destination processes can deliver all the messages sent to them in a round before the end of the round is reached. Suppose a multicast is initiated by a process at a time T (on its local clock). Reliability and ordering guarantees can be provided as follows: a multicast is either delivered at each functioning process at time $T + \Delta$ or is never delivered at any functioning process (atomic delivery); when $T + \Delta$ is reached (according to process local clocks) multicast messages timestamped with T will be ordered (in a deterministic way) before being delivered. This scheme (depending on how large Δ is) can have an bad effect on performance, since messages can only be consumed when the end of the round is reached, even if all the messages are received at the beginning of the round. Nevertheless, the termination time ($T + \Delta$) of multicast for synchronous systems is essential for applications requiring timeliness guarantees such as in real-time applications.

The assumption of bounded transmission delay and processing time (i.e. synchronous systems) can be realised in some systems settings by using special mechanisms such as running protocol processes with high-priority, guaranteed network bandwidth to "urgent" traffic, etc. However, the synchronous system assumption is quite hard to realise in system settings such as long-haul networks interconnecting general purpose multi-task systems, subjected to factors like:

[1] These protocols are actually refered to as Atomic Broadcast Protocols. Broadcast usually means sending a message to all nodes connected to a network. Multicast protocols, on the other hand, aim a sub-set of the nodes connected to a network and, more generally, a set of distributed processes. Thus, multicast can be thought of as a generarization of Broadcast. Therefore, the principles presented in the Broascast protocols cited can be directly applied to the corresponding multicast protocols.

unpredictable variations in system loads, queuing delays, unpredictable routings, message retransmissions due to errors, etc.

We are interested in group communication protocols for distributed systems with arbitrary network topology and diverse computing systems. Therefore, we have chosen to model our system as asynchronous, where no bound on message transmission delay and relative process speed can be accurately estimated. The no-bound assumption has a strong implication on failure detection, since under these circunstances it is impossible to distinguish between a crashed (or disconected) process and a slow one. On the other hand side, if a process does crash, it is important that non-faulty processes forming a group come to an agreement on that crash and remove the crashed process from the group; this is however impossible to achive within a finite time [Fischer85]. Asynchronous protocols therefore need to circumvent this impossibility result by permitting processes to *suspect* process crashes [Chandra91, Schiper93c] and to reach agreement only among those processes which they do not suspect to have crashed [Ricciardi91]. A process detected to have crashed will then be removed from the group membership of the functioning members involved in the agreement. Of course, there is the possibility of a functioning process erroneously be taken as crashed. Other implications of failures in asynchronous systems are discusses in section 2.5.

2.2 The System Model

We model our system as a set of sequential processes, distributed possibly in distinct processors or sites, which in their turn are linked by communication channels. We assume that processes can communicate to each other only by exchanging messages. We also assume the existence of a message transport layer permitting uncorrupted and sequenced message transmission between a sender and the destinations processes[2]. A group is defined as a collection of distributed processes in

[2]This functionality is provided by standard IPC mechanisms such as Unix TCP/IP sockets.

which a member process communicates with the other members only by multicasting to the full membership of the group. A given process can be member of more than one group. That means, we assume groups may overlap. No assumption about message transmission time is made (i.e. asynchronous modeling). Process and communication failures assumptions are discussed in section 2.5. In figure 2.1, we illustrate the system model. Applications are built on top of the group communication layer and the group communication layer is built on top of the transport layer.

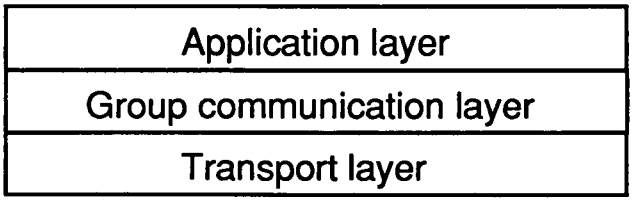


Figure 2.1 - The System Model

2.3 Overlapping Process Groups

In some applications, processes are required to simultaneously participate in multiple groups [Birman91c, Garcia-Molina91]. That is, process groups may overlap. Overlapping groups imposes extra complexity for multicast protocols since message delivery guarantees for different groups must take the existence of common members into account. In section 1.1.3 we have discussed the need for group overlapping in conferencing computer based applications. Now, we illustrate the use of overlapping groups by taking the problem of on-line server migration.

Suppose a given server is replicated in two processes, located in machines A and B, forming the process group g_1 (the upper-case letters represent the site name and also the replica process running in it). Also, suppose that the process in machine B has to migrate to machine C (figure 2.2(a)). This problem can be solved by using group overlapping in the following way.

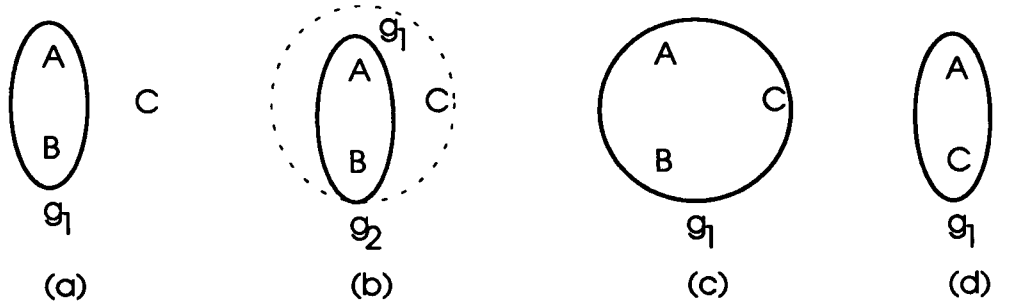


Figure 2.2 - On-line Server Migration

(i) Processes A and B agree that they will form a new temporary group g_2; from this point onwards, requests arriving to g_1 will be forwarded to g_2;

(ii) The new temporary group g_2 is formed that includes the new replica in machine C (figure 2.2(b));

(iii) The group g_1 is closed (i.e. it ceases to exist) and the temporary group g_2 is now renamed g_1 (figure 2.2(c)).

(iv) Process B leaves group g_1 (figure 2.2(d)).

Note that by using the group overlapping scheme suggested above, clients do not need to stop sending new requests while the migration process is being performed. However, request messages arriving to g_1 after the execution of step (i) and before step (ii) is concluded, must be buffered to be forwarded after conclusion of step (ii).

2.4 Message Order Delivery

Some distributed applications require that messages multicast are delivered in some specific order. For instance, processes maintaining replicated data require updates to be received in identical order. By our transport assumption, transmitted messages from a given sender are received by the destination processes in the same order they were sent. Thus, if just one sender is transmitting messages to a group, messages would be delivered in the same order they were sent. When more than one process is sending messages, however, there is the possibility of messages being delivered in a different order in distinct recipient processes. For example, consider that the messages m and m' are sent by two different processes to group *g*. A given

process p belonging to g could receive m before m' whereas another process q also belonging to g could received m' before m. Thus, delivery order of m and m' in p and q would violate the identical order principle, required for instance, by data replication applications. General purpose multicast protocols should therefore provide users with some delivery order guarantees. In the next sections we will discuss message order guarantees that can be provided by group communication protocols.

2.4.1 Event Ordering in Distributed Systems

A process execution consists of a sequence of events, each event corresponding to the execution of an action by a process. Within a given process, events are naturally ordered by the sequence they happen. However, ordering events from distinct processes is not possible unless they execute communication actions among themselves. An example of a communication action executed by a process, say p, can be to send a multicast message, say m, to a group that is recorded in m as $m.g$. The corresponding event will be denoted as $send_p(m)$. Similarly, we denote the event of a process q, belonging to the group $m.g$, receiving m as $receive_q(m)$. Then, we can define a 'happened before' relation, denoted as '$\rightarrow$', on *send* and *receive* events in a given set of system events. Thus, when a, b, and c are three distinct events in a subset of system events, each referring to either *send* or *receive* events,

(i) if a comes before b in the same process, then $a \rightarrow b$. e.g., if $send_p(m)$ comes before $send_p(m')$, then $send_p(m) \rightarrow send_p(m')$;

(ii) if a is a $send_p(m)$ and b is $receive_q(m)$, then $a \rightarrow b$; and

(iii) if $a \rightarrow b$ and $b \rightarrow c$, than $a \rightarrow c$.

A message m will be said to have potentially *caused* m', if $send(m) \rightarrow send(m')$, and distinct messages m and m' will be said to be *concurrent* if neither $send(m) \rightarrow send(m')$ nor $send(m') \rightarrow send(m)$ is true. Hence, the relation '$\rightarrow$' establishes a partial order of events in a distributed system [Lamport78]. For notational simplicity, when m and m' are two distinct multicasts, $m \rightarrow m'$ will denote that $send(m) \rightarrow send(m')$.

When some specific delivery order is required, received messages may have to be retained for later delivery until certain ordering conditions are satisfied (otherwise, the delivery order stated may be violated). Thus, we need to define $delivery_p(m)$ as the event of delivering message m to process p. Based on *send*, *receive*, and *deliver* events, and on the '$\rightarrow$' relation defined above, we can state the ordering properties applied to multicast protocols. To simplify the explanation, let us assume for a while there are no failures on communication channels or processes. We will consider process and communication failures in section 2.5.

2.4.2 Identical Order Delivery

As assumed by our asynchronous system model, messages may arrive at destinations with arbitrary delays. Some applications, however, require that the messages sent to a group of processes are delivered in the same identical order. For instance, an implementation of distributed lock on a set of replicated servers will require that locks are delivered to the replicas in the same identical order. The following is the specification of identical order for multicasts in a process group g, where $send_i(m)$ means process p_i multicasts m.

> **Identical order delivery :** Consider events $send_i(m)$ and $send_j$(m'), p_i and $p_j \in$ g: $deliver_q(m) \rightarrow deliver_q(m') \Rightarrow deliver_s(m) \rightarrow deliver_s(m')$, $\forall$ q, $s \in g$.

When groups overlap, common members of the overlapped groups must be delivered messages in identical order, even if the messages have been originated by members of different groups. In the example given in figure 2.3, messages m and m' have been sent by processes r $\in$ g_1 and s $\in$ g_2, respectively. Notice that p may receive m before m' whereas q may receive m' before m. However, p and q being common members of g_1 and g_2 should be delivered m and m' in identical order. Identical order delivery guarantee for overlapping process groups is specified as follows.

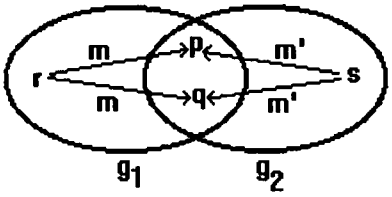


Figure 2.3 - Identical Order Delivery in Overlapping Groups

Identical order delivery : Consider events $send_i(m)$ and $send_j$(m'), $p_i \in$ m.g and $p_j \in$ m'.g: $deliver_q(m) \rightarrow deliver_q(m') \Rightarrow deliver_s(m) \rightarrow deliver_s(m')$, $\forall$ q, $s \in m.g \cap m'.g$.

2.4.3 Causal Order Delivery

Causality is a fundamental concept to many problems in distributed systems. For instance, determining a global snapshot of a distributed computation corresponds to finding a collection of local snapshots which is consistent with casual order [Chandy85]. For some applications, enforcing that messages are delivered respecting causal order may be essential to avoid inconsistences. For instance, suppose the existence of a replicated distributed data base, keeping bank accounts. Messages directed to the data base (account registers) correspond to deposits and withdrawals. Whenever an account balance is not enough for a withdrawal, the account value is not modified and a notice is given to the user about the attempt of withdrawal. Consider that the current state of an account X of user U is 0 units. Suppose that a process *p* sends a message *a* depositing 50 units in the account X, and afterwards *p* sends another message *b* withdrawing 40 units from the same account X. If *b* is delivered in a replica of the data base (a copy of the account register for X) before *a*, no fund will be available for that transaction and a notice will, inappropriately, be generated to U. If causal order is enforced, however, the messages deliveries will result in the balance of 10 units in all the replicas of X. Other examples of the use of causal order

protocols for a variety of problems in distributed systems can be found in [Birman89, Schmuck88].

If an event *a* "happened before" event *b* (i.e., a → b), then, event *b* may have been caused or influenced by event *a*. If *a* and *b* are two message multicasts, to respect causality, the delivery of *a* must precede delivery of *b*, in all common destinations of a and *b*. In figure 2.4, we illustrate causal delivery in a 3-member process group. The process group is formed by processes p_1, p_2, and p_3. The horizontal lines represent the passage of time and oblique lines message flows between processes (arrows touching the lines represent delivered messages). Process p_1 multicast message m_1 and process p_2, after delivering m_1, multicast m_2. Note that p_3 receives m_2 first but its delivery is delayed until m_1 is received and delivered.

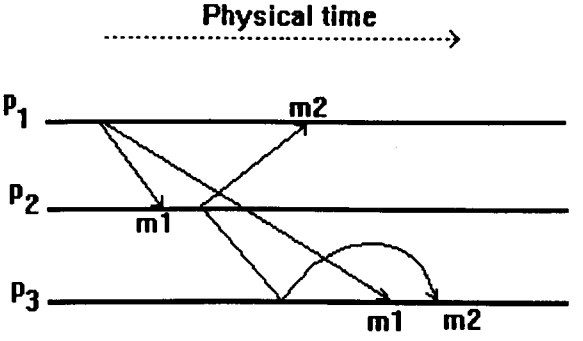


Figure 2.4 - Causal Order Delivery

If there is only one sender process transmitting messages to a group, sequenced delivered messages[3] (or FIFO delivery) would result in causal order always being respected. However, if more than one process are sending messages to the group, FIFO delivery alone is not enough to guarantee causal order delivery. Note that in the example given in figure 2.4, where p_1 and p_2 are transmitting messages, FIFO delivery would result in m_2 being delivered before m_1, leading to causal order violation. The specification of causal order delivery in a given group g is as follows.

[3] i.e. messages are delivered in the same sequence they have been generated by the sender.

Causal order delivery : $send_p(m) \rightarrow send_r(m')$, p and r $\in$ g $\Rightarrow$ $deliver_q(m) \rightarrow deliver_q(m')$, for any q $\in$ g.

When groups overlap, a message m received by a common member p, may actually be causally related to messages sent in other groups distinct from the one m has been sent to. In the example given below (figure 2.5), process p sends m_1 in group g_1. Process r, after delivering m_1, sends m_2 in g_2, and process q, after delivering m_2, sends m_3 in g_2 (i.e., $m_1 \rightarrow m_2 \rightarrow m_3$). In order to respect causality, message m_1 should be delivered to s before m_3 is delivered (notice that in the example of figure 2.5, s receives m_3 first). Below is the specification of causal order delivery in overlapping groups.

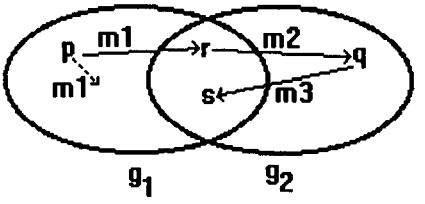


Figure 2.5 - Causal Order Delivery in Overlapping Groups

Causal order delivery : $send_p(m) \rightarrow send_r(m')$, p $\in$ m.g and r $\in$ m'.g $\Rightarrow$ $deliver_q(m) \rightarrow deliver_q(m')$, for all q, $q \in m.g \cap m'.g$.

2.4.4 Total Order Delivery

Total order delivery combines identical order with causal order delivery. Take the example used for causal order (a distributed data base keeping bank accounts). This time, consider that process *p* sends a message *a* with a deposit of 25 units and then a message *b* with a withdrawal of 50 units, both for account X of user U. Suppose also that a second process *q* sends a message *c* corresponding to deposit of 25 units for account X of user U. Notice that while messages *a* and *b* are causally related, message *c* is concurrent to both *a* and *b*. In this example, neither causal nor identical order, if applied in isolation, is enough to avoid inconsistencies. To verify this, assume, for

instance, that only causal order is enforced. Because concurrent messages are delivered in any order, we could have one replica getting message c before b (resulting a balance 0 for X) whereas other replica getting b before c (resulting in a notice issued to user U). Identical order in isolation also is not enough to avoid inconsistences. Note that the violation of causal order could lead to situations, where a deposit followed by a withdrawal by the same process, on behalf of a user, could result in an inappropriate notice for that user. So, we will need a total order message delivery. Identical order will prevent state divergences among the account replicas and causal order would avoid inappropriate notices being generated.

Total order delivery is an extension over the Causal order delivery, achieved by imposing a delivery order on concurrent events. Following is the specification of total order delivery for multicasts in overlapping process groups. Conditions (i) and (ii) assure that identical and causal order are not violated, respectively.

> **Total order delivery** : for any p_i, $p_j \in m.g \cap m'.g$: if $delivery_i(m)$, $delivery_i(m')$, $delivery_j(m')$, and $delivery_j(m)$ occur, then
>
> (i) $delivery_i(m) \rightarrow delivery_i(m') \Leftrightarrow delivery_j(m) \rightarrow delivery_j(m')$ and
> (ii) $send(m) \rightarrow send(m') \Rightarrow delivery_i(m) \rightarrow delivery_i(m')$.

2.5 Fault-Tolerance

We assume that processes fail by crashing. In the crash assumption, a process either works correctly (as specified) or stops working and from that point on does nothing. The process crash assumption separates cooperating processes into crashed and functioning ones. We assume the existence of a failure suspector [Chandra91, Schiper93c] which will suspect process crashes. In asynchronous systems, it is impossible to build a perfectly accurate failure suspector [Chandra91]. That is, failure suspectors may make mistakes. Erroneous suspicions made by failure suspectors can lead to virtual partitioning, where processes are sub-divided into functioning sub-groups unable to communicate with each other. A fault-tolerant multicast protocol

should therefore work correctly even when the original group has been partitioned into functioning sub-groups. Some existing systems however prevent the existence of multiple sub-groups [Birman91b]. In order to achieve that a group is allowed to exist as long as the majority its members are functioning, and by making (partitioned) processes not belonging to that majority leave the group.

To simplify the presentation of the fault-tolerant concepts, we will not consider in this chapter the existence of group partitioning. In chapter 6, we extend the fault-tolerant concepts to include network or virtual partitioning and present a protocol that can cope with that. In chapter 6, we also describe the implementation of a failure suspector based on local time-outs.

Let G_i be the set of groups a process p_i belongs to: $G_i = \{g_x \mid p_i \in g_x\}$. When p_i multicasts (or delivers) a message m with $m.g = g_x$, it actually does so only to (or from) those processes which it views as functioning members of group g_x. Let $V_{x,i}$ be the set of all processes which it views as functioning members of g_x. When g_x is initially formed, each functioning p_i installs an identical, initial view $V^0_{x,i} = \{ p_1, p_2, ..., p_n\}$. As p_i detects process crashes, it installs a new view by excluding the detected processes from its current view.

Processes failures can cause inconsistences among functioning members of a group. Firstly, if a multicast made by a process is interrupted due to the crash of that process, this can result in some connected destinations not receiving the message. Secondly, since views can be installed at arbitrary times, functioning processes may end up with different views of the group membership. For instance, two functioning processes may have delivered the same set of messages but have different views of the group membership. In some applications, group members require to have a mutual consistent view of the group membership changes, concerning all messages delivered. Consider, for instance, a load-sharing application where a distributed computation is divided up among the functioning members of a group. In this application, the task assigned to a group member will depend on state of the computation (messages delivered) and on the current group membership view. Therefore, in order to avoid

inconsistences (e.g., two processes being assigned the same task), group members are required be delivered the same set of messages for each group membership view installed.

In figure 2.6, we give an example of a 4-member group communication using only the functionality of our assumed transport layer and failure suspector. $V^0 = \{p_1, p_2, p_3, p_4\}$ is the group membership installed when the group is created and $V^1 = \{p_1, p_2, p_3\}$ is the new membership installed after the crash of process p_4 has been detected. Arrows touching the lines means message delivery and small black balls means new group membership views being installed. In this example, we illustrate two inconsistencies caused by the crash of process p_4. Firstly, message m2 was not delivered (and will never be) to process p_3. Secondly, message m1 was delivered to p_3 after it had known about p_4's crash, whereas p_1 and p_2 first deliver m_1 (and m_2) and then realize p_4 had crashed. We now proceed to precisely specify the properties that should be provided by fault-tolerant group communication protocols so as to avoid such inconsistences.

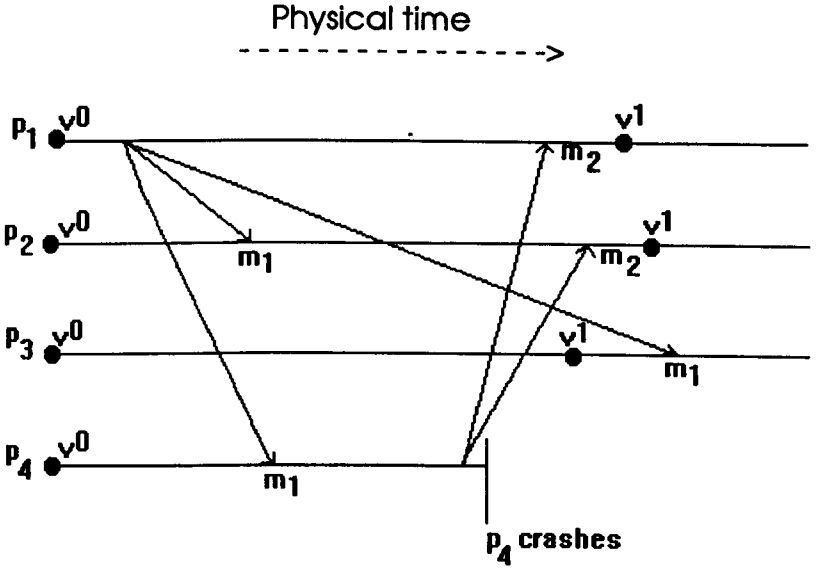


Figure 2.6 - Effects of crashes on Group Communication

Let $V^0_{x,i}$, $V^1_{x,i}$, $V^2_{x,i}$, ... $V^r_{x,i}$ be the series of views p_i has sequentially installed over a period of time, until it crashes (the view does not exist for a crashed process). The view updates by p_i must satisfy certain conditions so that message delivery by the protocol can be 'atomic' with respect to views updates. For this purpose, when failures occur (indicated by the failure suspector), functioning processes execute a membership protocol [Amir92b, Mishra91, Ricciardi92, Melliar-Smith91] to reach an agreement on the group membership of g_x. The view changes performed by processes of group g_x should satisfy the following properties:

VC1 : The sequence of views installed by any two functioning processes are identical (validity).

VC2 : If p_k crashes, $p_k \in V^r_{x,i}$ and p_i does not crash, then p_i will eventually install $V^{r'}_{x,i}$ such that $r' > r$ and $p_k \notin V^{r'}_{x,i}$ (liveness).

VC3 : any two functioning processes deliver the same set of messages between two consecutive views that are identical. That is, $V^{r-1}_{x,i} = V^{r-1}_{x,j}$ and $V^r_{x,i} = V^r_{x,j}$, then the set of m, $m.g = g_x$, delivered by p_i and p_j in V^{r-1}_x are identical.

VC1, *VC2*, and *VC3* are a formalization for the virtual synchrony model [Birman87, Schiper93a] which requires that (i) all functioning members "see" the same sequence of membership changes and (ii) for each membership configuration the same set of messages are delivered. *VC1* implies (i) and *VC3* implies (ii). *VC2* implies that if a process does crashes, the membership protocol will eventually generate a new view, removing the crashed process from the membership (in chapter 6 we will redefine these properties under the network partition assumption). Note that for the example given in figure 2.6, *VC1* is verified but *VC3* is violated. For instance, $V^0_1 = V^0_3$ and $V^1_1 = V^1_3$ but between these two identical views p_1 has delivered m_1 and m_2, whereas p_3 has delivered no messages.

2.6 Related Work

A number of group communication protocols have been described in the literature which meet many application requirements. Multicast protocols for synchronous distributed systems [Cristian90, Fischer83][1] are quite different in nature to the ones intended to the system model we are aiming, where no bounds on transmission and message processing delays are assumed. So, we will focus our attention on previous works intended for asynchronous distributed systems. The published protocols vary in several aspects: the availability of group management services (e.g. dynamic group membership service), the level of reliability provided (faults tolerated), the existence of distinct order delivery guarantees (e.g. causal, identical, total) and the techniques used to implement them. The research presented in this thesis builds on previous work which we will outline briefly in this section. In the subsequent chapters, when appropriate, we will make detailed comparisons of our work with solutions published to date.

2.6.1 Chang and Maxemchuk's protocol

In [Chang84] a fault-tolerant identical order protocol is described. Chang and Maxemchuk's protocol uses a sequencer process called the token holder (one of the group members) to generate sequence numbers for identical order message delivery. The token holder positive acknowledges messages sent to the group and multicasts the message sequence numbers, that will be used by group members to identically order the messages before delivery. Figure 2.7 illustrates the scheme of the Chang and Maxemchuk's protocol in a n-member group communication. In order to tolerate the failure of k members, delivery of a message is delayed until the token has been transferred to k distinct group processes. The message sequence numbers are also

[1] In fact, most of the work for synchronous systems are intented to broadcast (rather then multicast) messages to the whole set of processors (rather than a set of processes) connected to a network. Alhough problems such as group overlapping have not been addressed in broadcast protocols, the basic ideas present in those protocols can be directed applied to multicasts.

used by destination processes to detect lost messages that must be retransmitted by the token holder when required. Failure of the token holder is detected when a process does not receive the positive acknowledgement or when the token holder does not respond to a retransmission request. When the failure of the token holder is detected a process enters in a reformation phase when a new token holder is determined. Causal order and overlapping groups are not addressed.

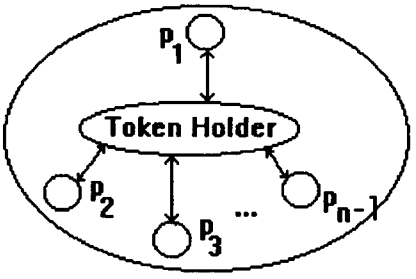


Figure 2.7 - Chang and Maxemchuk's protocol

2.6.2 V System and Amoeba

V System [Cheriton85] and Amoeba [Kaashoek91] are examples of systems which provide group communication protocols at the operating system level. V System group primitives perform best-effort to deliver a message to a group of processes. However, no guarantees are provided on the messages being delivered to all functioning members (atomic delivery[2]) or on the order messages are delivered. Amoeba provides an atomic and identically ordered group delivery primitive that works only for non-overlapping groups. Message multicasts are first passed to a sequencer process using point-to-point communication. The sequencer, after assigning a sequence number to the message, transmits it to the group. The sequencer waits for a number of acknowledgements from receivers before accepting that message for delivery. Membership (including the election of another sequencer)

[2] atomic delivery implies that if a functioning group process delivers a message, all functioning members of the group will also deliver that message. Property *VC3* (section 2.5) leads to atomic delivery.

reconfiguration is obtained by calling the ResetGroup primitive when a functioning process detects the failure of some member (indicated by the kernel). The protocol described in [Navaratnam88] has a similar approach to Amoeba. It is built on top of V System and provides identical order message delivery by means of a sequencer process (the primary manager) which gets all message broadcasts and delivers them in proper order to the other members. To achieve atomic message delivery, the primary manager waits for acknowledgements from all receivers (the secondary managers) before sending the next broadcast.

2.6.3 ISIS protocols

ISIS [Birman87, Birman91b] was the first system that implemented causal and total order protocols. The causal order ISIS protocol, called CBCAST, is based on the concept of vector clocks [Fidge91, Mattern89]. A vector clock is a n-dimensional vector where n is the process group size. Each process maintains a vector clock. The vector clock $vc(p_i)$ maintained by process p_i is initialized with zeros when p_i starts execution. When p_i multicasts a message, $vc(p_i)[i]$ is incremented by 1 and this value is transmitted as the message timestamp. When a message m timestamped with vc(m) is delivered by a process p_j, it updates its vector clock as follows :

$$\text{for } 1 \leq k \leq n,\ vc(p_j)[k] = \max(vc(p_j)[k], vc(m)[k]).$$

Assume vc1 and vc2 are two distinct vector clocks. We can compare vc1 and vc2 using the following rules (i) and (ii):

(i) $vc1 \leq vc2$ iff $\forall\, i : vc1[i] \leq vc2[i]$;

(ii) $vc1 < vc2$ iff $vc1 \leq vc2$ and $\exists\, i$ s.t. $vc1[i] < vc2[i]$.

Let $m \parallel m'$ denote the absence of causal relation between two messages m and m'. Also, when two vector clocks vc1 and vc1 cannot be compared by rules (i) and (ii) , we will denote $vc1 \parallel vc2$. Consider the relation 'happened before' defined by Lamport and denoted as '$\rightarrow$', and the messages m and m' timestamped with vector clocks. Causal relationship between m and m' can be precisely deduced by comparing their respective vector clock timestamps. That is, we can state the following property:

$m \rightarrow m' \Leftrightarrow vc(m) < vc(m')$ and $m \parallel m' \Leftrightarrow vc(m) \parallel vc(m')$.

A discussion on vector clocks properties can be found in [Raynal82].

Let vc_j be the vector clock maintained by process p_j. Assuming messages are transmitted in FIFO order, CBCAST delays the delivery of a message m sent by process p_i at a process p_j until $vc_j[k] = vc(m)[k] + 1$, for $k = i$, and $vc_j[k] \geq vc(m)[k]$ for $1 \leq k \leq n$ and $k \neq i$. The former test guarantees that previous messages from p_i have already been delivered to p_j and the latter test guarantees that the other messages sent to the group and delivered by p_i before it sent m, have already been delivered to p_j. In figure 2.8 we illustrate causal order delivery using vector clocks. Vector clocks values maintained by processes are in bold. Note that m_2, when received, is not immediately delivered at process p_3 because $vc_3[1] < vc(m_2)[1]$ ($vc_3 = <0,0,0>$ and $vc(m_2) = <1,1,0>$). When p_3 receives m_1 and delivers it, vc_3 becomes <1,0,0> and m2 can finally be delivered. After the delivery of m_2, vc_3 becomes <1,1,0>. Similar protocols for causal order delivery for point-to-point communication, rather than group communication, are presented in [Schiper89, Raynal91].

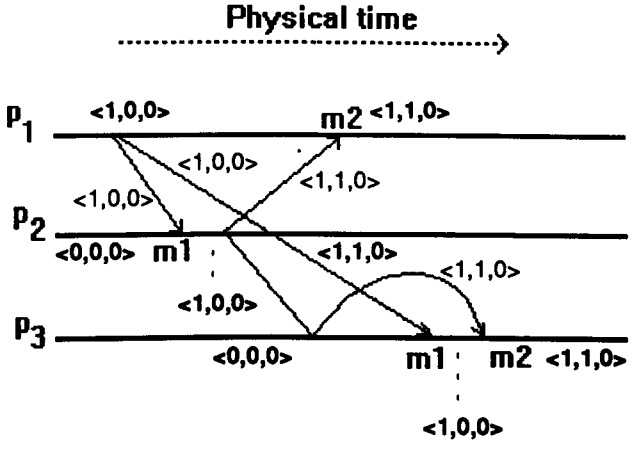


Figure 2.8 - Causal Order Delivery using Vector Clocks

A vector clock is a uni-group information. For the case of overlapped groups, CBCAST will carry with each transmitted message vector clocks of all groups involved in the overlapping structure. Depending on the number of groups, as well

their sizes, the transmission of vector clocks can produce a high message space overhead and this is the main drawback of CBCAST.

ISIS builds a total order protocol called ABCAST on top of CBCAST. Messages to be totally ordered are transmitted by CBCAST and kept marked "undeliverable" in a buffer. A special receiver (a sort of coordinator or sequencer process) called the token holder will multicast to the group the ordering information that should be used to deliver the buffered messages in total order [Birman91b].

Membership change consistency is guaranteed in ISIS by implementing the concept of virtual synchrony. So, in addition to causal or total order delivery, if the membership of a process group changes during a multicast, the semantics of virtual synchrony implies that the message will be delivered either to the members that were in the group before the change or to those that were in the group after the change. In other words, multicasts are delivered in the same membership view at all functioning members of a group [Birman89, Schiper93a].

2.6.4 Psync protocol

Psync [Peterson89] uses the concept of a conversation between processes. A process can start a conversation, join an existing one, send or receive messages to a conversation, or close a conversation. Messages sent to the conversation are delivered in causal order and, in order to achieve that, Psync [Peterson89] builds a directed acyclic graph to explicitly represent the causal relationship from all messages exchanged between processes during a given conversation. In the so called context graph, edges will link messages related by the Lamport's happened before relation such that the edge (m,m') will only exist if there is no path with size larger than one linking m and m'. Figure 2.9(b) is the context graph built from the example given in figure 2.9(a). Note that although $m_1 \rightarrow m_4$, the edge (m_1, m_4) is not represented in the context graph for there exist the path (m_1, m_2, m_4). Also notice that because m_3 and m_4 are not causal related, there is no directed path linking them.

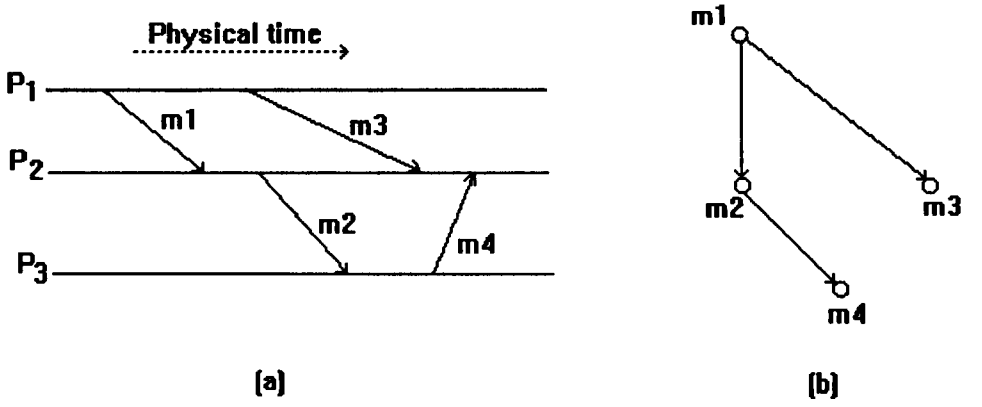


Figure 2.9 - Representing Causal Information in the Context Graph

Psync maintains a copy of the context graph at each host where there is a process participating in the conversation (copies of the context graph in different hosts are not necessarily identical at a given moment in physical time). When a message is transmitted to the conversation, besides being represented in the local context graph, the ids of all its predecessors in the local context graph are also sent together with the message (for instance, in figure 2.9, m_4 is transmitted with the id of its predecessor, m_2). When that message is received, it will be represented in the context graph of the remote destination process but delivery will only happen if all its predecessor messages are also present in the remote copy of the context graph. Otherwise, the message is put in a holding queue until its predecessor messages are received.

The basic send operation in Psync addresses only one process. So, a multicast primitive can be built by sending a message to all processes belonging to a conversation (a process group). A total order protocol can then be developed on top of Psync by identifying sets of concurrent messages and delivering messages of each set in a pre-fixed order [Peterson89]. This can be done by each group member incrementally executing a topological sort of the context graph (messages assigned to a given topological level - called a wave - are concurrent). In the example of figure 2.9, there are three waves : wave1 = $\{m_1\}$, wave2 = $\{m_2, m_3\}$, and wave3 = $\{m_4\}$. A wave is complete when a member process is certain that no future message will arrive that belongs to that wave (i.e., each participant has either sent a message to that

wave or sent a message that follows, in the causal relation, a message in that wave). In the example given in figure 2.9, wave1 and wave2 are complete. Wave3 is incomplete because process p_1 could still send a message which would fit in the topological level corresponding to wave3. In order to guarantee that waves will be complete, acknowledgements are sent for received messages (when the process has no application message to transmit). Process group overlapping has not been addressed in Psync.

2.6.5 Trans and Total protocols

In [Melliar-Smith90] two protocols are described for ordered broadcast[3] communication (Trans and Total) that are built on top of a hardware broadcast facility for local area networks. In order to guarantee that all transmitted messages are eventually received by all functioning processors, and also to construct a partial order on received messages, the Trans protocol uses a scheme of negative and positive acknowledgements plus the evaluation of an "observable predicate for delivery". The total order protocol named Total, incrementally extends the partial order provided by Trans on a total order. This is done by gradually selecting sets of candidate messages and by having some voting criteria deciding on the sets to be included in the total order. The positive and negative acknowledgements are piggybacked in transmitted messages. So, no extra control messages (other than null messages when processors are inactive) are transmitted. However, a considerable amount of processing time is needed in order to construct the partial and total orders out of the negative and positive acknowledgements. The Total protocol may not be able to decide on a order to deliver messages when processor failures occur. Nevertheless, the author claims that total order is achieved with high probability.

[3] a message broadcast is sent to all processors connected to a network rather than to a set of distributed processes.

2.6.6 Transis protocols

Transis [Amir92a] system provides the following multicast primitives: Causal, Agreed, Basic, and Safe. They implement causal order, total order, unordered with delivery atomicity, and unordered with uniform[4] delivery atomicity, respectively. As in Trans, the Transis system assumes the existence of a network broadcast facility and works by piggybacking negative and positive acknowledgements in transmitted messages. To represent causal information, Transis maintains a Directed Acyclic Graph (DAG) that is similar to the context graph of Psync. Transmitted messages carry acks for all direct causal dependent messages represented on the DAG graph. Transis also provides a membership service that guarantees the membership changes and message deliveries will happen in a consistent order at all processors (virtual synchrony semantics). The total order protocol of Transis [Dolev93] is built on top of the causal order protocol by applying functions on new messages added to the DAG graph that will eventually transform candidate messages into totally ordered messages.

2.6.7 Garcia-Molina and Spauster's protocol

In [Garcia-Molina91] is presented an interesting asymmetric (sequencer based) protocol that delivers messages in identical order in a group overlapping scenario. In this protocol, messages are ordered by a collection of processes (instead just one process as in [Chang84]) structured as a graph, called the propagation graph, which is built according to a particular set of multicast groups. Common members of overlapped groups are used as intermediate nodes of the propagation graph (a set of trees) so that messages from different groups can be delivered in identical order to multi-group members. A message is first sent to a primary destination and then propagated thorough the propagation graph until the intended destinations are reached. The main problem with this solution is that processes have to access the propagation graph which is a global piece of information and have to be reorganised

[4] Uniform atomic delivery implies that if a group process (whether functioning or not) delivers a message, all functioning processes of the group will also deliver that message.

due to dynamic group membership changes (due to process crashes, voluntary leaves, or joins).

2.6.8 Mostefaoui and Raynal's protocol

The protocol presented in [Mostefaoui93] provides causal order message delivery for overlapping process groups. For enforcing causal order delivery within a given process group, each group member multicasts messages following a scheme of phases. For each phase a process multicasts exactly one message, and the next phase begins when messages sent in the previous phase have been delivered. Message multicasts are timestamped with the phase number. When a process p receives a message timestamped with a phase number β for which it has not sent a message yet, it synchronises itself with the group by multicasting a null message containing the phase number $\beta + 1$. Thus, the group will be informed that p has progressed up to phase $\beta + 1$. In order to assess the phase progress for all the members, a process maintains a vector with one entry per process, say q, keeping the last informed phase number from q. A received message timestamped with t will be delivered at a process p only when p does not expect messages with timestamps equal or larger than t from any of the group members (this is concluded by consulting the vector of expected phase numbers). In order to cope with overlapping groups, a process p maintains a vector of expected phase numbers per group p may be a member of. Let max_X be the maximum expected phase number of processes of group g_X. Thus, when a process p transmits a message, it will be timestamped with the phase number max_Y for each group g_Y p belongs to as well as with the max_Z of other groups g_Z p has indirectly been informed by other members. A received message m will only be delivered at a process p, when all groups p is a member of, have progressed their phases up to the values indicated in the m's timestamp. This scheme produces a smaller message space overhead than the ISIS' CBCAST protocol for overlapping groups. CBCAST timestamps each message with one vector per group whereas Mostefaoui and Raynal's protocol timestamps messages with one integer value per group. CBCAST is however

more efficient in terms of delivery delay since vector clocks do not impose any extra synchronisation delays for message delivery.

2.6.9 Outlined Solutions and the Protocols developed in this Thesis

Chang and Maxemchuk's protocol, Amoeba, and the protocol presented in [Navaratnam88] address identical order for non-overlapping groups, using a sequencer process (a distinctive group member) to generate message sequence numbers for identical order. The ISIS' ABCAST protocol provides total order message delivery for non-overlapping groups and also makes use of a sequencer process. Garcia-Molina and Spauster's protocol extends in some sense Chang and Maxemchuk's protocol to work in overlapping groups. Processes using Garcia-Molina and Spauster's protocol will have to maintain the propagation graph that is built based on the group overlapping structure. Modification on the group memberships (due to process crashes or application related reasons) will lead to modification of the propagation graph, making it expensive to maintain for systems where groups change dynamically and frequently. In this thesis we develop a total order protocol (see chapter 4) that does not rely on sequencer processes such as in the propagation graph of Garcia-Molina' and Spauster's protocol. ISIS, Total, Transis, and Psync[5] provide total order message delivery (for non-overlapping groups) but their protocols have been built on top of their respective causal order protocols and require a larger message space overhead (the amount of ordering information added to application messages) when compared with our total order protocol. This is (mainly) due to the fact that our protocol has been built directly to provide total order (rather than being built on top of an existing causal order protocol). In order to assess delivery conditions, Total, Transis, and Psync protocols have to examine some 'stability' conditions in the structures they maintain for representing causal relationship between exchanged messages (negative and positive acks, the DAG graph, and the context

[5] In fact, Psync provides a poit-to-point causal order protocol. However, a total order multicast protocol can be built on top of this basic primitive as discussed in [Peterson89].

graph, respectively). In our total order protocol, this is done just by keeping the minimum value of a vector of integers and comparing this value with timestamps of received messages (integer values). Total and Transis total order protocols rely on the existence of a hardware broadcast facility. None of the protocols discussed addresses the requirement of total order delivery for overlapping process groups which has motivated the construction of our total order protocol.

The ISIS, Transis, Psync, and Mostefaoui and Raynal's protocols provide causal order delivery. However, overlapping groups are only addressed in ISIS and Mostefaoui and Raynal's protocols. ISIS provides an efficient causal order protocol called CBCAST. The CBCAST protocol delivers messages efficiently regarding message delivery delays. However, when complex group overlapping structures are considered, CBCAST can lead to a large message space overhead : a transmitted message has to carry vector clocks of all the groups from which messages have been exchanged (this will be examined in more details in chapter 5). In this thesis, we present three causal order protocols for overlapping groups with different trade-offs between message space overhead and message delivery delays. The first of our causal order protocols favours message space overhead (small one) with a longer message delivery delay. The second protocol favours message delivery delay but with a larger message space overhead. The third protocol is a compromise solution between message delivery delay and message space overhead, producing a trade-off similar to Mostefaoui' and Raynal's protocol.

In this thesis, we also describe fault-tolerant mechanisms for our protocols that make them to work correctly even if process failures and network partitions occur. Finally, the protocols and services we develop do not rely on any specific system facility such as a hardware broadcast.

2.7 Conclusions.

In this chapter we have discussed the main issues concerning the design of group communication protocols. We started by discussing the implications of synchronous

and asynchronous modelling on group communication. We have presented the system model adopted in the thesis and discussed issues related with group communication protocols (group overlapping, message ordering, and fault-tolerance). We have outlined existing group communication protocols and also pointed out some of their drawbacks, relating them with the protocols we develop in this thesis. In chapter 3 we will present the concept of Causal Blocks for maintaining ordering information for group communication. Causal Blocks representation will then be used to develop our protocols that we describe in the subsequent chapters (4 and 5). In chapter 6 we will show how fault-tolerance is incorporated to our protocols.

Chapter 3 - The Causal Blocks Model : Basic Principles and Concepts

The Causal Blocks model is a framework for developing group communication protocols with different ordering requirements [Macedo93a]. It also provides a mechanism for crash suspicion and membership reconfiguration. In this framework, communication (i.e., sending and delivery of messages) is thought of as proceeding in rounds where each round is characterised by a number of messages that hold no causal relationship. However, messages in different rounds may be causally related; so a round is termed a Causal Block. Each Causal Block is monotonically numbered and this number essentially represents a 'tick' from a logical clock [Lamport78], with the difference that Lamport's logical clock 'ticks' on *send/receive* events, whereas Causal Block numbers can be generated on a combination of events that may also include *delivery* of messages. By combining different ways of block numbering as well as different message timestamp contents (all based on Causal Block numbers, though), one can get distinct message ordering qualities. Moreover, the Causal Blocks representation provides an easy way of dealing with overlapping process groups. Integrated mechanism such as crash suspicion and membership reconfiguration (see chapter 6.0) have also been developed within the Causal Blocks context.

In the following sections of this chapter we first present the system model, we then describe how Causal Blocks are constructed, their basic properties, and associated concepts.

3.1The System Model and Failure Assumptions

We assume a set of sequential processes which are distributed possibly on distinct processors or sites and communicate with each other only by exchanging messages. We assume a transport layer that provides lossless, uncorrupted, and sequenced

message delivery between any pair of functioning processes (FIFO assumption)[1]. Once a message has been sent by the transport layer, it can take an potentially unbounded amount of time to be received at the destination. For simplicity, in this chapter we assume a failure-free environment (i.e., processes do not crash or misbehave) and we also assume that a given process can only belong to one given group (i.e., there is no overlapping of group memberships). A group once created will remain with the same membership until it is closed. Coping with dynamic changes in the membership (due to process crashes) is treated in chapter 6.0 and different ways of dealing with overlapping groups will be presented in chapters 4.0 and 5.0.

A *group* is just a set of cooperating processes. Members of a group communicate only by multicasting to the full membership of the group. A process execution consists of a sequence of events, each event corresponding to the execution of an action by a process. An example of an action executed by a process, say p, can be to send a multicast message, say m, to a group that is recorded in m as $m.g$. The corresponding event will be denoted as $send_p(m)$. We denote the event of a process q, belonging to the group $m.g$, receiving m as $receive_q(m)$, and $deliver_q(m)$ will denote the event of m being delivered to q. We distinguish the event of receiving a multicast message from the event of delivery, since in our protocols, as in other protocols, e.g. [Schiper89], a received message may have to be delayed before delivery, in order to satisfy some ordering condition, and also, as previously noted, because in the Causal Blocks model, deliver events may affect the block numbering process.

Throughout this chapter we will assume that all the members of a group are lively: a member will eventually multicast a new message. This assumption is put here only for didactic purposes and will be removed in chapter 4.0 with the introduction of the *time-silence* mechanism.

[1] This can be realised by introducing sequential numbers to messages, with positive acknowledgement and retransmission of missing messages (TCP/IP provides such a functionality).

3.2 Block Counters

Consider the existence of a group, $g = \{p_1, p_2, , \ldots , p_n\}$. Each process p_i maintains a logical clock called the Block Counter and denoted as BC_i. BC_i is an integer variable and its value increases monotonically. When g is created, the BC_i of every p_i is initialized to any finite integer, and without loss of generality, we will suppose that they are all initialized to zero. Transmitted messages are timestamped with Block Counters, and, as it is the case in Lamport's Logical Clock, timestamping using Block Counters will respect causality. That is, if $m \rightarrow m'$, then the timestamp associated with m' will be larger than the timestamp associated with m. However, the reverse will not always hold. If two messages have different timestamp values, they may be concurrent. Unlike Lamport's Logical Clock that is advanced on send/receive events, we have decided to advance Block Counters on send/delivery events, since a transmitted message can only be causally dependent on previously delivered ones. By proceeding this way, we reduce the number of messages with distinct timestamps that are concurrent. Specifically, we avoid the situation when a number of messages have been received (but not delivered) that would advance the local Block Counter, making the timestamp of a subsequent transmitted message larger than those timestamps of the mentioned received messages, where, clearly, they do not hold any causal dependence. More precisely, consider the *happened before* Lamport's relation denoted as '$\rightarrow$' (see section 2.6). When messages are delivered to processes using a multicast protocol, an occurrence of the following sequence of events is often possible in a process execution: { ... *receive*(*m*) $\rightarrow$ ... *send*(*m'*) ... $\rightarrow$ *deliver*(*m*) ... }[2]. Though *receive*(*m*) $\rightarrow$ *send*(*m'*), *receive*(*m*) cannot have caused or influenced *send*(*m'*), because *send*(*m'*) $\rightarrow$ *deliver*(*m*) and only delivered messages are used by a process in its computation. So, our *happened before* relationship representing the potential causality between messages will be defined based only on *send* and *deliver* events in a given set of system events. Thus, when *a*, *b*, and *c* are three distinct events in a subset of system events, each referring to either *send* or *deliver* events,

2 We will drop the suffix in denoting an event, where the suffix is obvious or irrelevant.

(i) if a comes before b in the same process, then $a \rightarrow b$;

(ii) if a is a $send_p(m)$ and b is $deliver_q(m)$, then $a \rightarrow b$; and

(iii) if $a \rightarrow b$ and $b \rightarrow c$, than $a \rightarrow c$.

Just before a process p_i multicasts a message m, it advances BC_i by one. The contents of the incremented BC_i is assigned to m as its block-number in the message field m.b. The Causal Blocks properties stated in this section will be equally valid when BC_i is incremented by any non-zero positive other than one (this will be the case in an overlapping scenario as we shall see in chapter 4). BC_i may also be advanced by p_i on $delivery_i(m)$ if the current value of BC_i is less than m.b. Thus, the two events under which BC_i may be advanced are:

CA1 (Counter advances during $send_i(m)$): Before p_i multicasts m, it increments BC_i by one, and assigns the incremented value to m.b; and,

CA2 (Counter advances during $delivery_i(m)$): Before pi delivers m, it sets $BC_i = \max\{BC_i, m.b\}$.

Based on *CA1* and *CA2* we can state the three following properties possessed by block-numbers of multicast messages. For notational simplicity, we denote $send_{pi}$ as simply $send_i$.

pr1 : $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$.

pr2 : for any m, $P_j \in m.g$: $deliver_j(m) \rightarrow send_j(m') \Rightarrow m.b < m'.b$; and,

pr3 : for all $m', m'' : m'.b = m''.b \Rightarrow m'$ and m'' are concurrent.

The properties *pr1* and *pr2* follow directly from *CA1* and *CA2*, respectively. Together they imply that for any distinct m, m': send(m) $\rightarrow$ send(m') $\Rightarrow m.b < m'.b$. The property *pr3* states that distinct messages multicast with the same block-number are necessarily concurrent and these messages must have been multicast by distinct processes, as *CA1* forbids two send events to occur in a given process with the same value of BC.

Finally, notice that the reverse of *pr1* is not always true. That is, if there are two distinct messages m and m' such that $m \rightarrow m'$, not necessarily $m.b < m'.b$ is true. To verify this statement, just notice that a sequence of send events will increment the Block Counter irrespective of the existence of received (but not delivered) messages. For instance, suppose that all messages with block-number B have been delivered in process p_i, and also that p_i has not sent or received any message with block-number B + 1 (i.e., $BC_i = B$). Now, suppose p_i receives a message m_0 with block-number B + 1 and then subsequently sends other two messages, m_1 and m_2. That is, it executes the following sequence of events: receive(m_0), send(m_1), and send(m_2). By *CA1*, $m_1.b = B + 1$ and $m_2.b = B + 2$. Although m_0 is concurrent to both m_1 and m_2 (note that deliver(m_0) has not happened), $m_0.b = m_1.b < m_2.b$.

3.3 Causal Blocks and the Block Matrix

Consider a process group $g = \{p_1, p_2, \ldots, p_n\}$. Using *pr3*, p_i constructs Causal Blocks to represent concurrent messages it sent/received with the same block-number. Construction of Causal Blocks leads to the notion of Block Matrix which can be viewed as a convenient way of representing sent and received messages with different block-numbers. A Causal Block is a vector of size equal to $n = |G|$. Whenever a process p_i receives or sends a multicast message with a new block-number, say B, it creates an empty vector of length n; for any message multicast with block-number B, it sets the i^{th} entry of the vector to '+'; and, for any multicast message received with block-number B from another process p_j, $j \neq i$, it sets the j^{th} entry of the vector to '+'. Causal Blocks, maintained by a process in this way, will have the following property:

PR1 : in a given Causal Block, only concurrent messages are represented.

A given Causal block is constructed to represent messages sent/received with the same block number, which by *pr3* are concurrent.

When group communication is active, i.e. member processes continually send multicast messages, the number of Causal Blocks constructed will grow. Causal Blocks maintained by a process are arranged in the increasing order of the message

block-number they represent, giving rise to a matrix which is called the *Block Matrix* and denoted as BM. Thus, BM[B] will represent the Causal Block for message block-number B. Referring to BM of any member process in G, we can state another property of Causal Blocks,

PR2: if m and m', represented in BM[B] and BM[B'], respectively, are causally related such that $m \rightarrow m'$, then $B < B'$.

If $m \rightarrow m'$, then either (i) the process which sent m also sent m', or (ii) the process that sent m' had previously delivered m. Suppose p_i and p_j are the processes which sent m and m', respectively. Suppose also that m.b = B and m'.b = B'. If (i) happens, i = j, and there will be a causal chain $send_j(m) \rightarrow ... \rightarrow send_j(m')$. After $send_j(m)$, by *CA1*, BC_j will be equalized to m.b. If (ii) happens, $i \neq j$, and there will be a causal chain $send_i(m) \rightarrow deliver_j(m) \rightarrow ... \rightarrow send_j(m')$. As in (i), after $deliver_j(m)$, by *CA2*, BC_j will be equalized to m.b. Since BC_j is never decremented (*CA1* and *CA2*), the block-number assigned to m' during $send_j(m')$, by *CA1*, will be larger than m.b.

PR2 also implies that if two messages are represented in different Causal Blocks, the one with smaller block-number may causally preceed (i.e., have caused) the message with larger block-number.

	p_1	p_2	p_3	p_4	p_5	p_6
1	+			+		
2		+			+	+
3	+		+	+		
4	+				+	
5		+				

Figure 3.1 - The Block Matrix of a 6-member Group Process

Figure 3.1 shows the Block Matrix of a 6-member process group. It represents all messages sent/received by the process which owns this particular matrix. Since

messages arrive at destinations with presumably distinct delays, the BM matrices of different processes will not necessarily be the same in a given moment of physical time. The BM matrix showed in figure 3.1, indicates, for example, that the block-numbers of the last messages received from processes p_1 and p_2, are 4 and 5, respectively.

3.4 Block Completion

By *PR1*, all messages which belong to a given Causal Block, say BM[B], are concurrent. From *PR2*, any causal order message delivery based solely on block numbers (see section 3.6) will require that a received message represented in BM[B], B > 1, be delivered only after all multicast messages which can be represented in BM[B'], for all B' < B, have been delivered.

To enable a member process to accurately determine that a given block completely represents all messages which can be represented in it, we use the notion of *block completion.*

A Causal Block BM[B], B ≥ 1, will be said to be *complete*, if and only if for all j, 1≤j≤ n, the j^{th} entry of BM[B] either (i) has '+' or (ii) is blank and there exists B', B'>B such that the j^{th} entry of BM[B'] has '+'. A message sent by process p_j and represented in the Block Matrix accordingly with (i) and (ii) is said to be the *completive* message from the group member p_j for Causal Block BM[B]. We can also say that a Causal Block B is complete, if there is a *completive* message for B, represented in BM, from each of the group members. By (i), it is ensured that if p_j has sent a message with block-number B, it is received. By *CA1* and *CA2*, p_j will not produce two distinct multicast messages with the same block-number. Since messages from a given sender process are received in FIFO order, (ii) guarantees that p_j has not multicast any message with block-number B. In the example shown in figure 3.1, block 2 is complete because processes p_2, p_5, and p_6 have sent a message with block-number 2, and processes p_1, p_3, and p_4 have sent a message with block-number 3.

From *PR2* it is obvious that the existence of BM[B], B > 1, also implies the existence of some Causal Block BM[B'], for some B' < B (B' = B - 1 if Block Counters are incremented by 1). Since messages are assumed to be transmitted in FIFO order, Causal Blocks in the BM maintained by a process will complete in the sequentially increasing order of block-numbers they represent.

In the next section, we introduce a compact representation of BM that provides an efficient way for detecting complete Causal Blocks.

3.5 The Last Received Vector - LRV

Each process maintains a vector of size n called the Last Received Vector, LRV, which contains the last received message block-number from each of the group members. In other words, consider the BM matrix maintained by a group member p_i, and that LRV_i denotes the LRV vector maintained by p_i. At any time, $LRV_i[j]$ will contain the largest block-number B of messages received from p_j (i.e. the j^{th} entry of BM[B] has a '+'). Let max{LRV} and min{LRV} be the maximum and minimum values present in LRV. Thus,

> *PR3 :* referring to the BM of pi, all Causal Blocks with block-numbers less or equal than $\min\{LRV_i\}$ are complete, and the number of incomplete blocks is given by the difference $\max\{LRV_i\} - \min\{LRV_i\}$.

The Causal Block BM[$\min\{LRV_i\}$] is complete because for every j, $1 \leq j \leq n$, there exist at least one Causal Block BM[B], $B \geq \min\{LRV_i\}$, such that the j^{th} entry of BM[B] = '+'. Since Causal Blocks get complete in the sequential order of their block-numbers, the Causal Blocks with block-numbers from 0 to (min{LRV} - 1) are also complete. Also, notice that any Causal Block BM[B], $B > \min\{LRV_i\}$, cannot be complete because there exist at least one entry j of BM (the one corresponding to one of the group members whose last received message block-number is equal to $\min\{LRV_i\}$) such that there is no '+' sign represented in BM[B]. Thus, the number of incomplete blocks is given by the difference max{LRV} - min{LRV}. In the example showed in figure 3.1, LRV= (4, 5, 3, 3, 4, 2), min{LRV} = 2, and max{LRV} = 5.

Causal Blocks 1 and 2 are complete, and there are 3 incomplete blocks (3, 4, and 5). Figure 3.2 illustrates the use of the LRV vector to determine the sets of complete and incomplete blocks. In the implementation of the protocol described in chapter 7.0, the LRV vector has been used together with other data structures to efficiently implement the BM matrix.

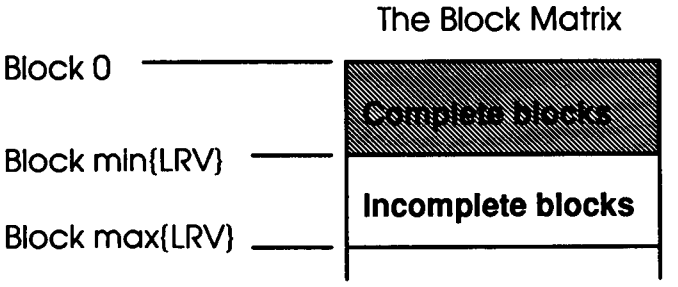


Figure 3.2 - Representation of Complete and Incomplete Blocks using LRV

3.6 Message Ordering

Message delivery protocols with different ordering requirements can be developed based solely on the concept of block completion. The following are safe conditions for causal and total order message delivery based only on block-numbers associated with transmitted messages and complete blocks in BM. Notice that we consider here that the Block Counter is incremented by one (section 3.2).

Conditions for Causal Ordering[3]

CO1 : messages with block-number 1 can be delivered soon after being received, and

CO2 : messages represented in BM[B], B > 1, can be delivered after BM[B-1] is complete and all messages represented in BM[B-1]have been delivered.

[3] Notice that message block-numbers do not precisely represent causal relationship between messages (the reverse of pr1 is not always true - section 3.2). In fact, CO1 and CO2 are delivery conditions which just do not violate causal order.

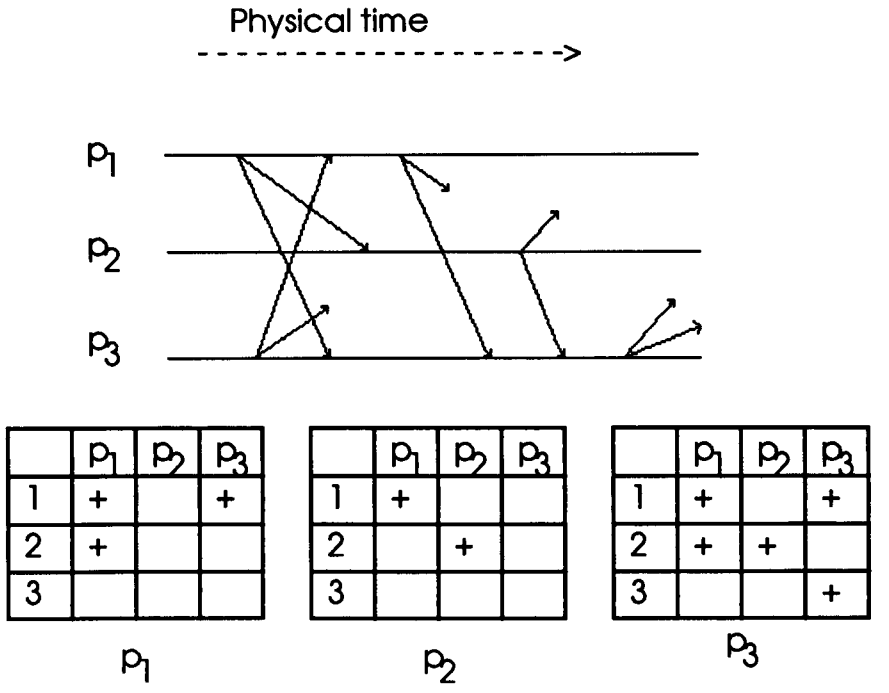
Conditions for Total Order

TO1 : messages represented in a given Causal Block will be delivered in a fixed pre-determined order, only after that block is complete, and

TO2 : messages in BM[B] will be delivered only after those in BM[B-1] have been delivered.

By applying these conditions, protocols satisfying either causal or total order message delivery can be developed.

Figure 3.3 shows the Causal Block Matrices of a 3-member group, at a given moment in time, when messages are delivered in causal order. Above the Block Matrices, the horizontal lines represent the passage of time and oblique lines message flows between processes (arrows touching the lines, represent received messages, and not touching the lines, represent messages in transit). In the BM of process p_3, blocks 1 and 2 are complete, and 3 is incomplete. Note that for block 3 to be complete, it would need messages from p_1 and p_2 with block-number equal or greater than 3.



	p1	p2	p3
1	+		+
2	+		
3			

p1

	p1	p2	p3
1	+		
2		+	
3			

p2

	p1	p2	p3
1	+		+
2	+	+	
3			+

p3

Figure 3.3 - The Block Matrices of a 3-member Group

3.7 Representing Missing Messages in the Block Matrix

So far we have assumed a sequenced, uncorrupted multicast transport layer. However, if the underlying message transport layer is taken to be unordered and unreliable, messages may not be received at the destinations, due to message loss or corruption. To detect missing messages, every multicast message, say m, is given not only a block-number, as *m.b*, but also the block-number, as *m.pb*, which was given to the message that was multicast prior to sending *m.b* by the same source. Similarly, when a message is received, it is not only represented in BM but also the representation of *m.pb* for the same sender is also checked. If a '+' sign does not represent *m.pb*, a '-' sign is placed to indicate that a message with block-number *m.pb* is expected, but not (yet) received due to out of order message delivery, or due to message corruption or loss. Loss of messages and out of order message delivery introduce some problems that are solved as explained below.

Firstly, when a '-' symbol is entered in the BM, the appropriate message source should be requested to retransmit the expected message. Secondly, the definition of complete blocks should be revised since the previous one was presented under no

failure and FIFO message delivery assumptions. A Causal Block BM[B], B ≥ 1, will be complete if and only if for all j, 0 ≤ j ≤ n, the jth entry of the Causal Block either has '+', or is blank and there exists a B', B' > B, such that BM[B'] = '+' and the j^{th} entry of BM[B"] is not '-' for all B", B < B" < B'. In other words, if any j^{th} entry of a Causal Block is blank, it will be followed by a (possibly empty) sequence of blanks and at least one '+' sign in the jth column of the BM. Note that due to '-' sign in the BM, BM[B] can be complete, while BM[b], b< B, is not complete. By the definition of Causal Blocks, a complete BM[B] will imply that all messages with block-number B have been received.

In the BM shown in figure 3.4, the first Causal Block is complete; but the second and the third blocks are not; however, the fourth block is complete. The second block is not complete because the third entry in it is a blank, followed (immediately) by a '-' sign in the third column. Though only a message from p_3 with block-number 3 is expected, it is not sure whether p_3's message with block-number 2 is also missing. The existence or non-existence of such a message can only be confirmed after the message with block-number 3 is received from p_3.

	p_1	p_2	p_3	p_4
1	+		+	+
2		+		
3	+		-	
4		+	+	
5	+			+

Figure 3.4 - Example of a Block Matrix in a 4-member Group

3.8 Conclusions

This chapter has presented the basic principles and concepts related with Causal Blocks. In describing these basic principles and concepts, we have made strong failure and membership assumptions that will be removed in the subsequent chapters. It has

been presented how Causal Blocks are created and become complete. It has also been presented conditions for causal and total order message delivery based solely on message block numbers. We finished the chapter showing how the sequenced, uncorrupted, and reliable transport layer assumption can be removed by introducing the representation of missing messages in the Block Matrix. In the next 2 chapters (4.0 and 5.0), we will present two protocols based on the principles we have introduced, extending the Causal Blocks concepts to work with multiple process groups and to precisely represent causal relationship between transmitted messages. In chapter 6.0 we show how fault-tolerance can be implemented in the context of Causal Blocks, removing the failure-free assumption made in this chapter. Chapter 7.0 then, describes the implementation of a total order delivery protocol based on Causal Blocks.

Chapter 4 - The Total Order Message Delivery Protocol, Newtop

Following the basic principles given in the previous chapter, we now present a total order message delivery protocol for overlapping process groups, also called the Newtop protocol (Newcastle total order protocol) [Macedo93b]. In this chapter, we also introduce a liveness mechanism called the *time-silence* mechanism, that will allow us to remove the assumption made in the previous chapter that processes are permanently active, sending messages.

An identical order protocol delivers messages in the same order in each of the destination processes of a group. A total order protocol, besides delivering the messages in identical order, also respects causality [Birman91b, Peterson89, Amir92a]. Previous works on total/identical order group communication protocols for asynchronous systems can be broadly classified into two categories: asymmetric and symmetric protocols. Asymmetric protocols usually rely on a sequencer process to which messages are first sent and then ordered for delivery [Chang84, Birman91b, Navaratnam88, Kaashoek91]. Symmetric approaches do not rely on any centralised process (such as a sequencer) and usually work by waiting for "sufficient" ordering information from group members before ordering received messages for delivery [Melliar-Smith90, Amir92a]. When processes are actively sending messages, the symmetric protocols work efficiently since message delivery will happen with the normal flow of messages (i.e., no extra control messages are needed). However, delivery delays can occur when members are inactive. To cope with this potential delays, inactive members are usually required to send periodic null messages. Asymmetric solutions have the extra communication cost with the centralised process (e.g. the sequencer) but will not delay message delivery due to inactive group members. Hence, if groups members may be inactive, asymmetric protocols are more efficient (no extra delays apart from transmitting the message to the sequencer). The

main advantage of symmetric solutions will come when failures and dynamic changes in the group membership (leaves and joins) are considered. Because symmetric protocols are fully distributed, the failure of a single group member will not affect the whole system (apart from the exclusion of the failed member from the group). When a sequencer fails, however, delivery of messages will be delayed until a new sequencer process has been established. Moreover, if groups are allowed to overlap, the choice of such a sequencer is a much harder task since it must be a common member of overlapping groups [Garcia-Molina91].

Newtop is symmetric and the message space overhead remains small (basically, the message block-number), despite the presence of multiple, possibly overlapped, groups. For the case of a single non-overlapping group in a failure-free environment, Newtop resembles the total order protocol presented in [Lamport78]. In some sense, we apply the concept of logical clocks developed in [Lamport78] to group communication context (i.e. multicasting rather than point-to-point communication), extending it to work with group overlapping and system components failures.

Section 4.1 describes the system model assumed in this chapter. Section 4.2 discusses how Causal Blocks are created for total order delivery. Section 4.3 presents the *time-silence* mechanism. Section 4.4 shows how overlapping groups are handled in Causal Blocks. Section 4.5 gives an algorithm description and proves the correctness of the total order protocol for overlapping groups. Finally, section 4.6 concludes the chapter and compares our solution with existing related protocols.

4.1 The System Model

As in section 3.1, in this section we will still assume a failure-free environment. However, two other assumptions not related with failures have been removed. Firstly, now we assume that processes can belong to multiple (overlapping) groups. Secondly, processes do not need to remain active by sending messages to guarantee that blocks eventually complete. The *time-silence* mechanism introduced in section 4.3 will work to guarantee that Causal Blocks created eventually complete.

4.2 Construction of Causal Blocks for Total Order Delivery

Newtop delivers all messages in the same order and that delivery respects causality. Formally, delivery of messages by Newtop satisfies the following property.

total order delivery[1]: for any p_i and $p_j \in m.g \cap m'.g$:

if $delivery_i(m')$, $delivery_i(m)$, $delivery_j(m')$, and $delivery_j(m')$ occur, then

(i) $delivery_i(m) \rightarrow delivery_i(m') \Leftrightarrow delivery_j(m) \rightarrow delivery_j(m')$ and

(ii) $m \rightarrow m' \Rightarrow delivery_i(m) \rightarrow delivery_i(m')$.

(i) guarantees that messages are delivered in the same global order and (ii) guarantees that delivery will respect causality.

To start with, consider the existence of a single group $g_X = \{p_1, p_2, \ldots, p_n\}$. It is assumed that every member process p_i, $1 \leq i \leq n$, knows the whole membership, and that, when g_X is created, the BC_i of every p_i is initialized to zero. The Block-Matrix of p_i for g_X will be denoted as $BM_{X,i}$. Thus, a message m sent or received by p_i, will be represented in the row $BM_{X,i}[m.b]$.

Before a process p_i multicasts a message m, it advances BC_i by α, where α, a non-zero positive integer, can be chosen randomly for any given multicast[2]. The contents of the incremented BC_i is assigned to m as its block-number in the field m.b. As BC_i is advanced by α, for every multicast, consecutively sent messages will have increasing block-numbers. Hence, the properties stated in section 3.2 are equally valid when BC_i is incremented by α. As stated in section 3.4, a Causal Block $BM_{X,i}[\beta]$, $\beta \geq 1$, will be said to be complete, when p_i can no longer send or receive a message, m, $m.b = \beta$ and $m.g = G_X$.

---

[1]Recall that we are assuming in this chapter a failure-free environment. In the presence of failures, total order delivery can not be guaranteed deterministically in an asynchronous system [Fischer85], unless mechanisms such as failures detectors [Hadzilacos93] are considered. Chapter 6.0 discusses such mechanisms in the context of Causal Blocks.

[2] The reasons for that will become clear when we discuss Causal Blocks properties for overlapping groups in section 4.4.

Recall that in section 3.2 we stated that both *send* and *deliver* events can advance the process Block Counters. For total order delivery, however, only send events will lead to the advancement of Block-Counters. The reason for that is simple. A message m can only be delivered to a process p_i, after the Causal Block $BM_{x,i}[m.b]$ is complete (*TO1* - §3.6). Note that the Causal Block $BM_{x,i}[m.b]$ can not be complete, if p_i has not sent a message with block-number larger than or equal to m.b. Thus, by the time $BM_{x,i}[m.b]$ completes, and its delivery occurs, BC_i would have been advanced to at least m.b, making *CA2* operation in section 3.2 redundant (i.e., $BC_i = \max\{BC_i, m.b\}$). *CA1* of section 3.2 is then, the only operation considered for total order delivery. Let us re-write the operation *CA1* in terms of α.

CA1 (Counter advances during $send_i(m)$): Before p_i multicast m, it increments BC_i by α, and assigns the incremented value to m.b.

When a process identifies complete Causal Blocks, it can identify the set of all concurrent messages with a given block-number (or the absence of it); also, the set of all messages that may be causally related to a given message can be identified; for instance, if a message m is represented in $BM_{x,i}[\beta]$ that is complete, then $\{m_0 \mid m_0 \rightarrow m\} \subseteq$ {messages represented by every $BM_{x,i}[\beta_0]$, $\beta_0 < \beta$}. Hence, Newtop obeys the following safe conditions for total order message delivery:

safe1: after a Causal Block is complete, a fixed pre-determined order for delivery is assigned to messages represented there; and,

safe2: a message m, $m.b = \beta$, is delivered only after the delivery of all messages with block-numbers less than β and the messages with block-number β that were ordered before m.

The above conditions state the safety property of Newtop for a single non-overlapping group. Its correctness follow straight from the properties *PR1* and *PR2* possessed by Causal Blocks and proved in section 3.3. Every received message will eventually be delivered by a process (liveness property), if it can be guaranteed that every Causal Block maintained by the process will eventually complete. This can only happen if every process is constantly generating messages and this was assumed in the

last paragraph of section 3.1. We now remove this assumption by introducing a simple mechanism called the time-silence mechanism, to enable a process to remain lively by sending null messages, during those periods it is not generating computational messages to complete created Causal Blocks.

4.3 Time-silence Mechanism

The time-silence mechanism of a process p_i, $timesilence_i$, works as follows: whenever p_i creates a new Causal Block as a result of receiving a multicast message with block-number β, a timeout for some predetermined period (called *local-time-silence*) is set for that Causal Block, $BM_{x,i}[\beta]$, if p_i has not already multicast a message with block-number larger than or equal to β. This timeout period indicates the duration within which p_i is expected to multicast a message with block-number β or larger - thus contributing to the completion of $BM_{x,i}[\beta]$ at all member processes of g_x (including itself). Note that p_i multicasting a message with block-number β' will contribute to the completion of blocks $BM_{x,j}[\beta_0]$, for $\beta_0 \leq \beta'$ and for all j, $1 \leq j \leq n$. So, if p_i multicasts a message with block-number β', $\beta' \geq \beta$, before the expiration of the timeout set for $BM_{x,i}[\beta]$, then the timeout set for any and every $BM_{x,i}[\beta_0]$, for $\beta_0 \leq \beta'$, are cancelled. If, on the other hand, the timeout for $BM_{x,i}[\beta]$ expires, then $timesilence_i$ will force p_i to multicast a special null message. This null message is multicast with the largest block-number that p_i has "seen" so far, i.e. with a block-number $\beta'' = \max\{m'.b \mid receive_i(m')\} \geq \beta$, so that, this multicast will contribute to the completion of all Causal Blocks $BM_{x,j}[\beta_0]$, for $\beta_0 \leq \beta''$ and for all j, $1 \leq j \leq n$. This null message will also cancel the timeouts set for any $BM_{x,i}[\beta_0]$, for $\beta_0 \leq \beta''$. With the introduction of time-silence, Block-Counters of processes advance not only by sending application related messages (*CA1*), but also when null messages are sent. This possibility is stated as *CA2*.

CA2 (Counter Advances due to sending of a null message by timesilence$_i$) before p_i multicasts a null message m, it sets m.b = max{m'.b | receive$_i$(m') has occurred}, and BC_i = m.b.

Null messages contains only protocol related information (such as block-number, group identifiers, etc.). They are distinct and distinguishable from application related messages, which will be called non-null messages, where distinction is required. Just like a non-null message, a null message, upon being received, is represented in BM_X; also, if the reception a null message creates a new Causal Block, the timesilence will start a timeout for that block, if a multicast with a larger or equal block-number has not already been made. A null message due for delivery, will not be supplied for processing.

Note that for a multicast, block-numbers are computed using different algorithms for null and non-null messages. Despite this difference, *prl* ($send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$) is still valid. That is, successive multicasts from p_i will have increasing block-numbers.

proof: if m' is non-null, *prl* is true by *CA1*, irrespective of whether m is null or non-null. Suppose then that m' is a null message. Irrespective of whether m is null or non-null, send$_i$(m) must have caused all unexpired timeouts set for any $BM_{X,i}[\beta_0]$, for $\beta_0 \leq$ m.b, to be cancelled. In other words, no null message m' (m' $\neq$ m) could have been sent by timesilence$_i$ with block-number m.b. Therefore, send$_i$(m') must have occurred because of p_i received a null or non-null message, say μ, with μ.b > m.b. Hence, by *CA2*, m'.b $\geq \mu$.b > m.b.

The time-silence mechanism can increase the message overhead of the protocol when processes are not always active. However, the time-silence mechanism or some equivalent mechanism (such as periodic exchange of 'I am alive' or 'synchronise' [Mostefaoui93] messages by processes) is essential for ensuing the liveness of any fully distributed (symmetric) total order protocol.

4.4 Overlapping Groups

Consider two groups $g_1 = \{p_1, p_2, p_3, p_4\}$ and $g_2 = \{p_3, p_4, p_5, p_6\}$. The processes p_3 and p_4 are members of both g_1 and g_2 (see figure 4.1). Suppose that p_1 multicasts m_1 in g_1 and that m_1 is delivered to p_3 which subsequently multicasts m_2 in g_2. P_4, being a member of g_1 and g_2, will receive m_1 and m_2 (not necessarily in that order) and must be able to deduce that $deliver_3(m_1) \rightarrow send_3(m_2)$, i.e. $m_1 \rightarrow m_2$. Newtop treats group overlapping in the following way: processes which are members of more than one group should maintain a single Block Counter which should be advanced subject to *CA1*, no matter which one of the groups a sent message belongs to. So, if the multigroup member process p_3 maintains a single BC for both g_1 and g_2, then the block-number given to m_2 will be $m_2.b > m_1.b$, as $send_1(m_1) \rightarrow deliver_3(m_1) \rightarrow send_3(m_2)$.

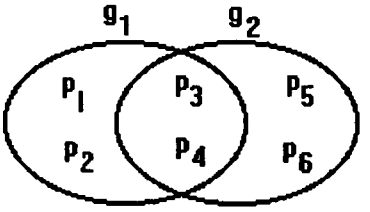


Figure 4.1 - Overlapping Groups $g_1$and g_2

More precisely, suppose that a process p_i is a member of more than one group. Let G_i be the set of groups p_i belongs to: $G_i = \{g_x \mid p_i \in G_x\}$. So, $|G_i| > 1$. We assume that p_i maintains, as before, a single BC_i which will be updated as mentioned in *CA1* - irrespective of the group m was multicast. It will, however, maintain a distinct $BM_{x,i}$ for each group g_x in G_i - representing messages sent or received with $m.g = g_x$ in the same manner as described in 4.2. Although p_i advances its BC_i by only one every time it multicasts a non-null message, it can appear to advance its BC_i by a randomly chosen α, $\alpha > 0$, between successive multicasts in a given group, if it has performed multicasts to other groups in between those two multicasts. Despite this, the following Causal Block property will still be valid in a given $BM_{x,i}$ of p_i: if two distinct non-null messages m and m' are represented respectively in $BM_{x,i}[\beta]$ and $BM_{x,i}[\beta']$, and $m \rightarrow m'$, then the row $BM_{x,i}[\beta]$ will precede the row $BM_{x,i}[\beta']$.

The time-silence mechanism of p_i will operate independently for each distinct g_X: the timeout set for $BM_{X,i}[\beta]$ will not be cancelled when p_i multicasts a μ with $\mu.b \geq \beta$ and $g_X \neq \mu.g$; similarly, if the timeout set for $BM_{X,i}[\beta]$ expires, a null message will be multicast only in g_X. As the working of time-silence mechanism and the multicasting of null messages become group-oriented, *CA2* and validity of *pr1* will become restricted to a given single group.

CA2': Just before p_i multicasts a null message m in a given group g_X, it sets $m.b = \max\{m'.b \mid receive_i(m') \text{ has occurred} \wedge m'.g = g_X\}$, and $BC_i = \max\{BC_i, m.b\}$. This will change *pr1* to:

pr1': $send_i(m) \rightarrow send_i(m') \wedge m.g = m'.g \Rightarrow m.b < m'.b$, for any m and m'.

Since the messages multicast by a given process in a given group have increasing block-numbers and are received in FIFO order, a multi-group member process p_i can identify the complete blocks in each of its $BM_{X,i}$ in the same manner as in section 3.4. Note that by *CA1*, $m \rightarrow m' \Rightarrow m.b < m'.b$, for any non-null m and m'.

To summarize, p_i maintains a Block Matrix for each of the groups it belongs to, and the time-silence mechanism of p_i operates for each group as if the other groups did not exist. By applying the arguments presented in non-overlapping case, it can be shown that any single, or multi-group member process in the system will eventually identify every Causal Block it creates to be complete.

The message delivery condition, *safe1*, must however be modified to take account of the fact that a process belongs to more than one group. The new condition for a process, p_i, is:

safe1': after $BM_{X,i}[\beta]$ is complete for every $g_X \in G_i$, a fixed pre-determined order for delivery is assigned to the messages with block-number β.

We illustrate the need for the above modification with the help of a simple example. Consider the case illustrated in figure 4.2(a), where m_1 and m_2 are messages with the same block-number (say β); naturally we require that p and q be delivered these messages in the same identical order, and before delivery of any message with

block-number larger than β. Assume that r has fast communication paths to p and q, and r follows m_1 with a few more multicasts in g1. Assume that while m1 has been received at p and q, m_2 is still in transit. If we use condition *safe1* (rather than *safe1'*), then p (q) could be delivered m_1 and its successor messages from r followed by m_2. Use of condition *safe1'* will prevent this from happening.

The time-silence mechanism ensures that, once a block has been created, it will eventually complete. However, this liveness mechanism alone is not enough to guarantee block completion for the case of overlapping groups. Referring again to figure 4.2(a), suppose s does not multicast m_2. How long should p (q) wait, before being absolutely certain that no message with block-number β is in transit in g_2 ? We solve this problem by adding another liveness mechanism that works as indicated below.

Whenever a process multicasts a non-null message, with say block-number χ, to one of its groups or receives a non-null message with block-number χ in a group, it also multicast a null message with block-number χ in those other groups whose Block Matrices do not contain a Causal Block with block-number larger than or equal to χ (thereby ensuring that Causal Block with block-number χ will eventually complete in each group).

Let us now return to the previous example. Suppose q receives m1, and its BM for g_2 contains entries whose block-numbers are less than β (β = m1.b); so q will send a null message with block-number β to members of g_2. This will guarantee that s will definitely create a Causal Block with block-number β and contribute to the completion of this block either by transmitting m_2, or a null message (generated by the time-silence mechanism of s for g_2 - $timesilence_{2,s}$). The message delivery conditions, *safe1'* and *safe2*, together with the time-silence and the added liveness mechanism can cope with arbitrarily complex group structures. Figure 4.2(b) illustrates a cyclic group structure. Assume that $m_1 \rightarrow m_2 \rightarrow m_3$, p does not send any further message to r, and m1 is still in transit as m_3 is received at r. Causal delivery at r ($m_1 \rightarrow m_3$) is guaranteed because of the following four reasons: (i) *safe1'* will

require Causal Blocks with block-number $m_3.b$ to be complete both at g_3 and g_2; (ii) the liveness mechanism will ensure that p does create a Causal Block with block-number $m_3.b$, and contribute with a null message; (iii) the null message from p will arrive at r only after m1 (transport layer FIFO assumption); and finally, (iv) *safe2* will ensure causal delivery.

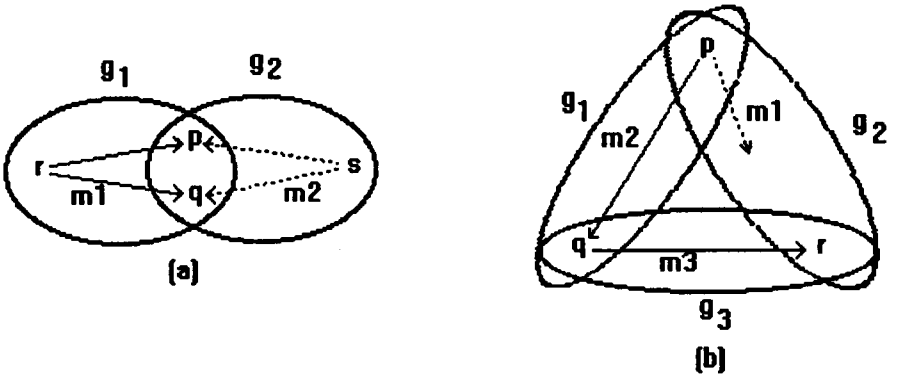


Figure 4.2 - Acyclic and cyclic Overlapping Groups

4.5 Protocol Description

4.5.1 Algorithm

Having given the basic principles for construction of Causal Blocks for overlapping groups, we proceed to present the algorithms of the relevant aspects of protocol[3], and in the subsequent section, to show its correctness arguments. We will assume that the global variables such as the Block Counter, Block Matrix, etc., are accessed by the various sub-processes that make the protocol, in a mutually consistent manner.

Consider process p_i, and let $G_i = \{g_x \mid p_i \in g_x\}$. To run the protocol, a process p_i has - in addition to BC_i and a $BM_{x,i}$ for each g_x in G_i - a vector, called the Completed Blocks Vector and denoted as CBV_i, which is of size $|G_i|$ and is maintained in the following manner: for each g_x in G_i, $CBV_i[x]$ indicates the largest

[3] An actual implementation in C++ using TCP/IP sockets is discussed in chapter 7.0.

number of complete blocks in $BM_{x,j}$. Process p_i also maintains the value of $\min\{CBV_i[x] \mid g_x \in G_i\}$ in a variable B_{min}. CBV_i is initialized to [0,0,...,0], B_{min} to 0, and every time a '+' is added to any $BM_{x,i}$ both are updated if possible. Whenever the value of B_{min} changes, all sent and received, but not delivered messages with block-number less than or equal to (the new value of) B_{min} are delivered according to the message delivery conditions *safe1'* and *safe2*. We will use the notation $max_{x,i}$ to represent the largest block-number in block-matrix $BM_{x,i}$.

We first describe the algorithm by which p_i multicasts a non-null message m to the destination group m.g. As stated before, p_i also multicasts a null message to those groups $g_x \in G_i$ where $m.b > max_{x,i}$. In order to distinguish the event of multicasting a message (null or non-null) used in the definition of the relation '$\rightarrow$', from the actions resulting from the multicast of a non-null message (including the extra null messages cited above), we term the latter set of actions as *psend(m)*. Accordingly, *preceive(m)* will denote the actions taken by *msg-receive* sub-process described below when receiving a message (we will use the symbols '$\rightarrow$' and '$\square$' for 'then' and 'else', respectively).

```
psend(m : non-null-message):
{
   m.b := BC_i + 1; BC_i = m.b;
   multicast m to other members of m.g;
   prepare a null message η with η.b = m.b;
   for each g_x in G_i - {m.g} →
    do
      m.b > max_x,i → η.g := g_x; multicast η to other members of g_x
      □ m.b ≤ max_x,i → skip
    od
}
```

Block-numbering and multicasting a null message in response to the expiration of timeout set by time-silence will proceed as per *CA2'*. We assume that multicasting a null or non-null message will automatically update the appropriate BM and will modify CBV_i if necessary. We will also assume a primitive to deliver a message whose execution will supply the message to the local destination process, only if that message is non-null. The protocol for p_i is given below, dropping the suffix i.

The *msg-receive* sub-process receives messages from the transport layer intended for p_i. When the received message is non-null, a null message is also multicast to those $g_x \in G_i$, where m.b > $max_{x,i}$. The sub-process *msg-deliver* awaits the value of B_{min} to change (or to increase, to be more precise). When the value of B_{min} changes, it delivers all sent and received, but not delivered messages with block-number less than or equal to B_{min}, in a fixed and pre-determined order.

```
begin
msg-multicast:

  { ... uses psend(m) ... }

||

msg-receive:
{ do
    receive(m); update appropriate BM; compute CBV;
    if m is null  → skip;
    ❒ m is non-null → G' := G - {m.g};
       for each g_x in G - {m.g} →
         do
           m.b ≤ max_x → G' := G' - {g_x};
           ❒ m.b > max_x → skip
         od
         if G' = { }  → skip
         ❒ G' ≠ { } → prepare a null message η with η.b = m.b;
                          BC := max{η.b, BC};
            for each g_x in G' →
              do
                 η.g := g_x; multicast η to other members of g_x
              od
         fi
    fi
  od
}

||

msg-delivery:
{ do
      if B_min = min{CBV[x] | g_x ∈ G} → skip
      ❒ B_min ≠ min{CBV[x] | g_x ∈ G} →
            B_min := min{CBV[x] | g_x ∈ G};
            deliver all sent and received, but not delivered m such that
             m.b ≤ B_min in the fixed, pre-determined order;
      fi
  od
}

||

time-silence:

  { // manage time-silence timouts }

end
```

4.5.2 Correctness of the Protocol

For proving the correctness of the protocol, we have to show that a sent message is eventually delivered (liveness property) and it does so without violating causal/total order (safety property). Before proving these properties, we will show that the sending of null messages by the *psend* and *preceive* primitives will not violate property *pr1'* (§4.4) based on which completeness of Causal Blocks is identified. We will also prove that for any non-null messages m and m', if $send_i(m) \rightarrow send_i(m')$, then $m.b < m'.b$.

observation1: Notice that either primitives (*psend* or *preceive*) after being executed, say for a message μ, will not decrease the current value of the Block-Counter whose value will be always equalized to the block-number of the message sent (if any is sent). That is, its value will be set to $\max\{BC_i, \mu.b\}$.

Lemma I : The sending of null messages by the *psend* and *preceive* primitives does not violate *pr1'* ($send_i(m) \rightarrow send_i(m') \wedge m.g = m'.g \Rightarrow m.b < m'.b$, for any m and m').

proof:

If m' is non-null, then *pr1'* is true by *observation1* and *CA1*, irrespective of whether m is null or non-null. Suppose then, that m' is a null message sent by p_i in a given group $g_y \in G_i$, $y \neq x$ (where $g_x \in G_i$ is the group corresponding to the non-null message, say μ, taken by either primitives). There are two cases to consider here. Firstly, if m' is a consequence of psend(μ), its block-number is the same of the corresponding non-null (i.e. $m'.b = \mu.b = BC_i + 1$) and that will be larger than m.b since, as stated in *observation1*, the multicast of m (either in psend(μ) or preceive(μ)) always keep BC_i to the $\max\{BC_i, m.b\}$. Secondly, m' may be a consequence of preceive(μ). In this case, m'.b is not the incremented value of BC_i but the block-number of the received message μ in g_x. Now, suppose by absurd hypothesis that there is a message m previously sent by p_i in g_y with $m.b \geq m'.b$ (it could be a null message sent by $time\text{-}silence_{i,y}$ or just another non-null message sent by p_i in g_y), so that $max_y \geq m.b \geq m'.b$. Then, the violation of *pr1'* is avoided by the statement present

in both *psend* and *preceive* primitives that forbids a null message such as m' being sent to those groups $g_y \in G_i$ where $m'.b \leq max_y$; that is, m' could not exist.

Observation2: Observe that because the block-number of the null messages generated by time-silence is calculated per group basis (*CA2'* - §4.4), *pr1'* is no longer valid when those messages are considered. For example, take the two block matrices for groups g_x and g_y a process p_i simultaneously belongs to. Assume p_i has not sent any message yet (i.e. $BC_i = 0$). Also, assume that p_i receives a message μ in g_x with block-number 4 and after some time another message μ', in g_y, with block-number 2. Suppose, the timeout for μ expires before the timeout for μ'. Thus, From *AC2'*, $timesilence_{i,x}$ will force p_i to send a null message, say η, $\eta.b = 4$, and BC_i will be set to 4. Similarly, $timesilence_{i,y}$ will force p_i to send another null message, say η', $\eta'.b = 2$, and this time BC_i will not be affected. Hence, although $\eta \rightarrow \eta'$, $\eta.b > \eta'.b$. Notice also, that in spite of the fact that block-numbers of null messages from time-silence are not monotonically generated, the value of BC_i never decreases. However what have been observed, when only non-null messages are considered (and only non-null can be consumed by processes), if $send_i(m) \rightarrow send_i(m')$, indeed $m.b < m'.b$.

Lemma II: $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$, for any non-null m and m'.

proof: if m.g = m'.g, the assertion becomes identical to *pr1'*, and by Lemma I that will be true for any m and m'. So, consider m.g ≠ m'.g. Just after $send_i(m)$, by *psend*, BC_i will be set to m.b. Since by *observation1* and *observation2*, BC_i is never decreased no matter which group p_i sends the message to, and *psend*(m') will increment BC_i before assigning its value to m'.b, indeed $m.b < m'.b$.

Safety property

We must show that messages are only delivered in the same global order and that causality is not violated. Consider two non-null messages m and m', such that p_i and p_j are in m.g ∩ m'.g. If send(m) → send(m') then m.b < m'.b (by *Lemma II*). Suppose that m.b < m'.b. As B_{min} never decreases, it is not possible for ($B_{min} \geq$ m'.b) to become true before ($B_{min} \geq$ m.b) becomes true. Since messages are delivered

in the increasing order of their block-numbers, deliver(m) $\rightarrow$ deliver(m') will be true for both p_i and p_j. Suppose that m.b = m'.b. For every process p_k in m.g $\cap$ m'.g, when ($B_{min} \geq$ m.b) becomes true for the first time, both $BM_{m'.g,k}$[m.b] and $BM_{m.g,k}$[m.b] will be complete; so, both m and m' will be taken together for delivery and their delivery order will be according to the pre-determined order which is fixed identically for all processes in the system.

Liveness property

We must show that a sent message will eventually be delivered at the destination processes. Firstly, notice that by our transport assumption, any sent message will eventually be received at the destination processes and that the *time-silence* mechanism guarantees that a created Causal Block (resulting from a received or sent message) eventually completes. A message m, represented in BM[m.b] of a process p_i, is only delivered after $BM_{x,i}$[m.b] is complete for every $g_x \in G_i$ (*safe1'* - §4.2) so as $B_{min} \geq$ m.b. Consider that m is non-null and that was multicast by p_i in g_x. The operation $psend_i(m)$ ensures that a null η with η.b = m.b is multicast such that in every $g_y \in G_i - \{g_x\}$, a $BM_{y,i}$[m.b] is created, if not already there. When p_j, $p_j \in g_x$, receives m, the *msg-receive* procedure also ensures that $BM_{y,j}$[m.b] is created (if not already there) for every $g_y \in G_j$. Since, as noted previously, every Causal Block created completes (by *time-silence*), and is identified to be complete, $B_{min} \geq$ m.b eventually becomes true and thus m is eventually delivered.

4.6 Conclusions

In this chapter, we have presented an approach based on Causal Blocks for implementing a symmetric total order protocol for groups that may be overlapping. Besides being simple to handle, even in the presence of overlapping groups (a process can belong to multiple groups), the approach presented has the main advantage of the constant and low message space overhead. We have also presented in this chapter, a mechanism to ensure that a created Causal Block will eventually complete. This has been called the time-silence mechanism.

Symmetric total order protocols have also been presented in [Melliar-Smith90, Amir92]. However, these protocols have not addressed the problem of multiple groups; further they assume in their system model a hardware broadcast facility on top of which the protocols are built (making them not as portable as Newtop). Asymmetric protocols with similar functionalities are described in [Chang84, Navarantnam88, Birman91, Kaahoek91]. None of them however, addresses the group overlapping problem for total order delivery. In [Peterson89] a symmetric protocol is described where group overlapping is not addressed either. Both total order protocols in [Birman91b and Peterson89] are built on top of a causal order protocol which make them have a larger message space overhead (the amount of ordering information added to user messages) when compared to Newtop: in [Birman91b] messages carry vector clocks and in [Peterson89] messages carry references to other messages related by the 'happened before relation' and represented in a graph, called the context graph. Newtop, in contrast, has been designed directly to provide total order delivery and messages are timestamped with just a block-number. In order to assess delivery conditions, the protocol in [Peterson89] has to examine some 'stability' conditions in the context graph. In Newtop, this is done just by keeping the minimum value of the vector CBV and comparing this value with the block-numbers of received messages. The protocol presented in [Garcia-Molina91] indeed addresses the group overlapping problem. However, all processes have to access the propagation graph (see section 2.6) which is a global piece of information and have to be reorganised due to dynamic group membership changes (due to process crashes, voluntary leaves, or joins). That is, the delivery of messages will be delayed until the graph is reorganised and its new configuration known by the relevant processes. In Newtop, only local information is used to handle group overlapping and changes in the overlapping structure will not affect message ordering.

The next chapter shows other protocols based on Causal Blocks and intended only for causal order delivery. In chapter 6.0, we incorporate fault-tolerance to Newtop, removing the failure-free assumption made in this chapter.

Chapter 5 - Causal Order Message Delivery in Overlapping Process Groups

If two messages m and m' are causally related (m $\rightarrow$ m'), a causal order protocol will deliver m' to a process p only after m has been delivered to p. If the delivery of m' is delayed only if there exist a message m", such that m" $\rightarrow$ m' and m" has not yet been delivered to p, we say that the causal order protocol delivers messages as early as possible. Causal order protocols have to compromise between delivery speed (as early as possible principle) and the size of message timestamps containing causality information [Mostefaoui93]. Small timestamps usually means slower delivery. The first causal order protocol for process groups, the CBCAST protocol [Birman87], was based on the idea of message piggybacking. In CBCAST, causally related messages were piggybacked on top of transmitted messages; this made the protocol quite expensive in terms of message space overhead. In the second version of CBCAST [Birman91b], a message to be transmitted is timestamped with a vector clock [Mattern88, Fidge91, Schiper89, Birman91b] which is a compact representation of a causal history within a process group. Vector clocks lead to a precise causal dependency representation between transmitted messages, and it is a lower-bound in message space overhead in order to get delivery as early as possible [Charron91, Raynal92]. When groups overlap, causal order protocols based purely on vector clocks [Birman91b] are quite expensive in terms of message space overhead. Typically, vector clocks related to all groups in the system have to be transmitted on top of transmitted messages. Although some optimisations are possible (e.g., using compression - [Birman91b]), the high cost is unavoidable to guarantee fast causal delivery (as early as possible delivery). In order to avoid the high cost of vector clocks for overlapping process groups, some compromise must be made, trading a potentially slower delivery with smaller timestamps.

Using conditions *CO1* and *CO2* of section 3.6 for causal message delivery, when a message m arrives at a destination process p, it can only be delivered after the Causal Block (m.b - 1) becomes complete in p. As we have previously stated, some of the messages with block-number less than m.b may actually be concurrent to m (the reverse of *pr1* is not always true - section 3.2). Although the waiting time introduced by block completion is inevitable for the total order requirement, it can be avoided for just causal order delivery, if we can deduce the precise set of messages m depends on. In this chapter, we shall show how Causal Block numbers can be used to precisely represent causal relationship between transmitted messages (section 5.1), leading to a faster causal order protocol. We will then present three protocols for causal order delivery in overlapping process groups, in the context of causal blocks[1], and describes a trade-off solution in more details. The first protocol, called Slow causal protocol, is purely based on message block-numbers. It is optimum in message space overhead but may introduce extra delay time for message delivery because it relies on block-completion. The second protocol, the Fast causal protocol is based on a vector called the Global Last Delivered Vector (GLDV). It reduces message delivery delay (delivery as early as possible) but has a higher message space overhead. The third protocol, the Relative causal order protocol, uses the Last Delivered Vector (LDV) combined with the block-completion concept and is a trade-off solution between message space overhead and delivery delay.

Sections 5.2 discusses causal order delivery in overlapping process groups. After presenting the system model and failure assumptions in section 5.3, we describe the Slow, Fast, and Relative causal order protocols, in sections 5.4, 5.5, and 5.6, respectively. Section 5.7 concludes the chapter.

1As messages are organized in Causal Blocks, all complementary mechanisms, such as membership and flow control (chapters 6 and 8 respectively) can also be used in the causal order protocols presented in this chapter.

5.1 Representing Causal Relationship precisely using Causal Blocks Numbers

Consider that instead of timestamping a message m with just its block-number, we introduce a timestamp that includes also the block-numbers of the last delivered message from each of the group members by the time m is sent. When m is received at a destination process p, it can be delivered immediately after the messages represented by those block-numbers have also been delivered at p.

Consider that each member process p_i of a group g maintains a vector, called the Last Delivered Vector and denoted as LDV, whose size is the size of g. The contents of LDV, together with the message block-number, will be used to timestamp sent messages. At any time, LDV[j] of p_i, $1 \le j \le n$, will indicate the largest block number of the message from p_j that p_i has delivered. Assume that m.b and m.s denote the block-number and the sender process identifier associated with message m, respectively. Thus, LDV of p_i will satisfy the following at any given time,

$$LDV[j] = \max\{m.b \mid m.s = p_j \text{ and } \textit{deliver}_i(m) \text{ has occurred}\}.$$

With respect to its LDV, p_i will carry out the following:

A1 : just before $\textit{send}_i(m)$: m.ldv = LDV,

A2 : just after $\textit{send}_i(m)$: LDV[i] = m.b (or BC_i), and

A3 : just after $\textit{deliver}_i(m)$: LDV[m.s] = m.b,

Consider the LDV vectors maintained by processes p_i and p_j, and denoted as LDV_i and LDV_j, respectively. The values of LDV_i and LDV_j are compared using the following rules:

C1 : $LDV_i \le LDV_j \Leftrightarrow \forall k : LDV_i[k] \le LDV_j[k]$

C2 : $LDV_i < LDV_j \Leftrightarrow LDV_i \le LDV_j$ and $\exists k : LDV_i[k] < LDV_j[k]$

For causal message delivery, a received message *m* is delivered to p_i immediately after the following delivery condition is true:

DC : for all $p_j \in m.g$, $m.LDV[j] \le LDV_i[j]$.

By the above condition, any message that p_i sends is immediately deliverable at p_i. Hence, we will assume that for *m*, *m.s* = p_i, $send_i(m)$ is immediately followed by $deliver_i(m)$.

LDV vectors have similar properties to vector clocks (see section 2.6.3). The difference is due to the fact that an entry of a given LDV represents the Block Counter value of a given group process (i.e. the process logical time), whereas an entry of a vector clock indicates how many events have happened in a given process by a given moment of its computation. In other words, $LDV_i[j]$ indicates not only the last message sent by process p_j and delivered by p_i, but also informs p_i that p_j has sent/delivered messages up to Causal Block $LDV_i[j]$. This p_i's knowledge of p_j's logical time progress, as we will show in this chapter, provides our protocol with an efficient way of dealing with causal message delivery in the presence of group overlapping. Below we state some properties of LDV vectors.

LDV vector Properties

Consider messages *m1* and *m2* timestamped as m1.LDV and m2.LDV, respectively. Based on the previous definitions we state the following properties *pr1*, *pr2*, and *pr3* possessed by LDV vector timestamps.

pr1 : $m1.b = \max(m1.LDV[j], 1 \leq j \leq n) + 1$;

When a message m1 is sent, its block-number m1.b is given by the incremented value of the current Block Counter (by *CA1* - *section 3.2*). Thus, to verify *pr1*, we only need to show that the value of a Block Counter $BC_i = \max(LDV_i[j], 1 \leq j \leq n)$. Notice that for i = j, $LDV_i(j)$ is only updated to equalise BC_i (by *A2*). For $i \neq j$, by *A3*, $LDV_i(j)$ is updated to m1.b of a delivered message m1. By *AC2* - *section 3.2*, BC_i will maintain the maximum value between the previous value of BC_i and m1.b ($LDV_i(j)$ after *A3*). Hence, $BC_i = \max\{LDV_i[j], 1 \leq j \leq n\}$ holds, and therefore, pr1.

pr2 : $m1 \rightarrow m2 \Leftrightarrow m1.LDV < m2.LDV$ and $m1.b \leq m2.LDV[i]$

Let p_i and p_j be the processes which sent m1 and m2, respectively. By the definition of the relation '$\rightarrow$', if $m1 \rightarrow m2$, then either i = j and m2 was sent after m1, or m2 was sent after P_j has delivered m1. In the former case, by *A2*, $LDV_i[i]$ will be

incremented after m1 be sent, thus m2.LDV > m1.LDV and m1.b ≤ m2.LDV[i] will hold. In the latter case, notice that m1 can only be delivered in p_j after LDV_j be larger or equal than m1.LDV. Because the block-number of a message is always larger than all block-numbers present in its timestamp vector (by *pr1*), then, after delivering m1, by *A3*, $LDV_j(i)$ is updated to m1.b and then will be larger than m1.LDV[i]. Therefore, by *A1*, the inequality m2.LDV > m1.LDV will hold. Because $LDV_j(i)$ is updated to m1.b before being used as m2's timestamp, then m1.b ≤ m2.LDV[i] will also hold. Hence, it follows that if m1 → m2, then m1.LDV < m2.LDV and m1.b ≤ m2.LDV[i]. The reverse of *pr2*, that is, if m1.LDV < m2.LDV and m1.b ≤ m2.LDV[i], then *m1* → *m2*, follows from the fact that when m2 is timestamped to be transmitted, all messages with block-numbers up to those present in m2.LDV will have already been delivered in p_j, including *m1* (note that *m1.b* ≤ *m2*.LDV[i]). Thus, if m1.LDV < m2.LDV and m1.b ≤ m2.LDV[i], then indeed *m1* → *m2*.

Notice that by using LDV vectors as timestamps, the potential causal relationship between messages is as precisely represented as in vector clocks (see section 2.6). In practice, this precise representation of potential causality makes possible fast causal message delivery. That is, the delivery of a message *m* is only delayed if there is another message m' such that *m'* → *m* and *deliver(m')* has not happened yet.

If two messages *m1* and *m2* are not causally related they are said to be concurrent messages, and we will denote m1 || m2. Concurrent messages can be precisely defined by their timestamps as follows:

pr3 : m1 || m2 ⇔ neither {m1.LDV < m2.LDV and m1.b ≤ m2.LDV[i]}
nor {m2.LDV < m1.LDV and m2.b ≤ m1.LDV[j]} holds.

If m1 || m2, neither m1 → m2 nor m2 → m1 hold. In the former case, when m2 was sent m1 had not been delivered yet. Thus, by *A3*, $LDV_j[i]$ will only progress to m1.b when m1 is delivered in p_j. Thus, m2.LDV[i] < m1.b. The same reasoning applies for the latter case (when m2 → m1 does not hold).

5.2 Causal Order in Overlapping Process Groups

When process groups overlap, a message m received by a multi-group member may causally depend on messages sent in other groups distinct from the one m was sent to. Thus, timestamping a message with a single group vector (such as LDV or Vector Clocks) is not enough to represent causal information. Consider two groups g_x and g_y, such that $g_x \cap g_y \neq \emptyset$ (i.e. g_x and g_y overlap). A message m sent in g_x by a process in $g_x \cap g_y$, may depend on previous messages sent in g_y. Thus, m has to carry causal information not only about g_x but also about g_y so that any other member of $\{g_x \cap g_y\}$ will be able to deliver m properly.

Overlapped groups can be represented as a graph of groups whose structure can vary in time due to the changes on the group memberships. Consider the existence of process groups $g_i = \{p_1, p_2, \ldots, p_n\}$. The graph O of overlapped groups is defined as follows: one vertice of O represents a given process group, and there will be an edge linking two vertices whenever the corresponding groups overlap. That is, O = (V,E), where V is the set of all groups g_i and $(g_j, g_k) \in E$ iff $g_j \cap g_k \neq \emptyset$. In the particular case where there is no cycles in O, it suffices for a message m to carry causal information about all groups the sender of m is a member of. However, in the general case, causal dependences may propagate through cycles [Birman91b] and messages have to carry global causal information (i.e., causal information of all groups in O). For instance, consider the 4 groups $g_1 = \{p_1, p_2, p_3, p_4\}$, $g_2 = \{p_3, p_4, p_5, p_6\}$, $g_3 = \{p_5, p_6, p_7, p_8\}$, and $g_4 = \{p_1, p_2, p_7, p_8\}$. The graph O of overlapping groups (figure 5.1) is given by $V = (g_1, g_2, g_3, g_4)$ and $E = \{e_1 = (g_1, g_2), e_2 = (g_2, g_3), e_3 = (g_3, g_4), e_4 = (g_1, g_4)\}$. Consider that p_1 sends the message m_1 in g_1. After delivering m_1, p_3 sends m_2 in g_2. Then, p_6 delivers m_2 and sends m_3 in g_3. And finally, p_7 delivers m_3 and sends m_4 in g_4. Since $m_1 \rightarrow m_4$, p_2 has to deliver m_1 before m_4. Unless the information '$m_1 \rightarrow m_2 \rightarrow m_3$' is passed through the groups to p_7, it cannot deduce '$m_1 \rightarrow m_4$' and send this causal information together with m_4.

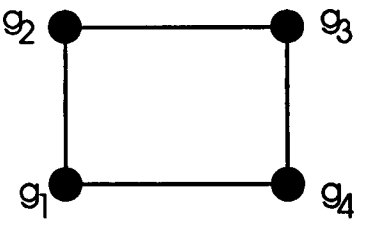


Figure 5.1 - Graph O of overlapped process groups

5.3 System Model and Failure Assumptions

We will assume the system model as specified in chapter 4, section 4.1. So, let us assume initially that processes do not crash so that once a group membership is installed it will not change during of the execution of the causal order protocol. Failures for the causal order protocols presented in this chapter will be treated in the same way as for the total order protocol Newtop. For this, refer to chapter 6.

Based on *send* and *deliver* events, and on the '$\rightarrow$' relation defined in 3.2, we state the following property satisfied by the protocols developed in this chapter.

Causal order delivery: $send_p(m) \rightarrow send_r(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$, for all functioning q, $q \in m.g \cap m'.g$, $p \in m.g$, and $r \in m'.g$.

5.4 The Slow Causal Order Protocol

We present the slow causal order protocol by slightly modifying the total order protocol described in chapter 4. In that protocol, a message m can be delivered only after the Causal Block BM[m.b] is complete. This is necessary so as the concurrent messages can be delivered in the same (pre-fixed) order. For just causal order, however, concurrent messages can be delivered in any arbitrary order. Therefore, a received message m can immediately be delivered, provided that all blocks BM[B], B < m.b, are complete and its messages delivered. The condition *CA2* of section 3.2 that was redundant for total order, now must be used to advance the block-counter after delivery. So, we take the total order protocol presented in chapter 4 and modify the *msg-delivery* sub-process (section 4.5.1) to produce just causal order delivery. Figure 5.2 shows the modified msg-delivery sub-process intended for causal order delivery.

```
msg-delivery: /* executed after a message has been received */
{
 do
  if B_min = min{CBV[x] | g_x ∈ G} →
     deliver all sent and received, but not delivered m such that m.b = B_min + 1
     if m.b > BC_i → BC_i := m.b; /* update the block-counter */
  ❑ /* the received message completed one or more blocks */
     B_min := min{CBV[x] | g_x ∈ G};
     deliver all sent and received, but not delivered m such that m.b ≤ B_min + 1
     if m.b > BC_i → BC_i := m.b; /* update the block-counter */
  fi
 od
}
```

Figure 5.2 - The msg-delivery sub-process for slow causal order delivery

In figure 5.2, B_{min} represents the largest block-number of complete (and delivered) Causal Blocks for all groups p_i is a member of. Thus, any received message with block-number $B_{min} + 1$ can immediately be delivered to p_i with the assurance that all possible causal dependent messages have been already delivered. On the other hand, messages with block-numbers $B_{min} + K$, $K > 1$, will have to wait for block-completion to occur. Otherwise, causal order may be violated. For instance, if a message m, $m.b = B_{min} + 5$, is received, it can not be immediately delivered since another message m', $m'.b = B_{min} + K$, $1 \leq K \leq 4$ and $m.s \neq m'.s$, can be subsequently received. Therefore, m will have to wait for the block $BM[B_{min} + 4]$ to become complete in order to be safely delivered. When more than one block get complete, messages are delivered following the increasing order of their block-numbers. Safety and liveness correctness proves for the Slow Causal order protocol are similar to the ones given for the total order protocol in chapter 4.

5.5 The Fast Causal Order Protocol

To obtain fast causal order delivery, we extend the group based Last Delivered Vector presented in section 5.1 into a Global Last Delivered Vector suitable for a multi-group environment. Consider the existence of distinct process groups. In order

to construct the GLDV vector, each process maintains a unique Block-Counter BC, despite the number of groups it may belong to. To represent sent/received messages, each process p_i maintains a Block-Matrix $BM_{i,x}$ for each group g_x that p_i is a member of. Updating BC_i follows the rules CA1 and CA2 (section 3.2). Also, each process p_i maintains a Global Last Delivered Vector ($GLDV_i$) for keeping the largest block-number of messages sent/delivered by all processes (belonging to groups p_i is a member of or not).

Suppose the existence of a multi-group process p_i. Thus, $GLDV_i$ is initialised with the entries corresponding to all processes belonging to groups p_i is also a member of. Transmitted messages are timestamped with the current value of $GLDV_i$. So, before p_i transmits a message m, it assigns the current value of $GLDV_i$ to m.gldv. Just after m is sent, the vector entry corresponding to p_i is updated with m.b (i.e. $GLDV_i[i] = m.b$). When m is received by another process, say p_j, the local $GLDV_j$ is updated to include all entries in m.gldv that are not present in $GLDV_j$. So, actually, $GLDV_j$ will maintain the largest block-number of messages sent/delivered by processes of groups in the overlapping structure O. To simplify presentation, let us assume that when a given GLDV is initialised, there will be an entry in it for each process in O and they will be set to zero before any message is transmitted or delivered. Notice that like the LDV vectors (section 5.1), the GLDV vectors are constructed by maintaining the largest block-number of messages sent/delivered by processes. Thus, the comparison rules and properties of LDV vectors presented in section 5.1 can also be applied for GLDV vectors. Consider a given process p_i, and let $G_i = \{g_x \mid p_i \in g_x\}$. Let S be the set of all processes belonging to groups p_i is also a member of. That is, $S = \{p_k \mid p_k \in g_x \text{ and } g_x \in G_i\}$. On receiving a message m, p_i must proceed with the following test for delivery:

Delivery test : if ($\forall\ p_k \in S$, $GLDV_i[k] \geq m.gldv[k]$) then deliver(m);

After delivering the message m, $GLDV_i$ is updated as follows:

Updating GLDV:

for any process $p_k \notin S$, $GLDV_i[k] := \max(m.gldv[k], GLDV_i[k])$;

$GLDV_i[m.s] := m.b;$

The delivery test guarantees that a received message m is only delivered after all messages represented in m.gldv, corresponding to those processes in S, have already been delivered by p_i. After delivery of m, the entry of $GLDV_i$ corresponding to the sender of m (m.s) is updated with the block-number of the message delivered (m.b). The other entries of $GLDV_i$ not corresponding to processes in S, will keep the maximum values between the values in $GLDV_i$ and m.gldv, so that a future transmitted message m' by p_i will carry in m'.gldv all inter-group causal information known by p_i by the time m' is multicast.

Message space overhead when using the GLDV vectors

Consider the existence of a set of process groups whose overlapping structure is given by the graph O, previously defined. The size of GLDV, |GLDV|, corresponds to the number of processes participating in groups in O. It can be trivially verified that $|GLDV| < \sum |g_i|$, $g_i \in V$, if $E \neq \varnothing$, where $|g_i|$ is the size of group g_i. In CBCAST [Birman91b], a message is timestamped with |V| vector clocks (i.e. the number of groups in O), leading to a message space overhead of $\sum |g_i|$, $g_i \in V$. Thus, the CBCAST[2] overhead corresponds to the GLDV upper-bound overhead, that will only happen when there is no overlapping in O ($E = \varnothing$). Hence, the more groups overlap, the smaller is the overhead of Fast causal protocol using GLDV when compared with the CBCAST.

5.6 The Relative Causal Order Protocol

We now combine the group based LDV vectors with the concept of block-completion to develop a trade-off solution. We will first describe the protocol for the non-overlapping scenario. We will then extend the protocol to work with overlapping process groups, proving its correctness.

2. Without considering the compression mechanisms presented in [Birman91b].

To represent ordering causal information between messages sent in a given group, we will make use of the Last Delivered Vector (LDV) introduced in section 5.1. Thus, a process p_i will maintain a Block Counter $BC_{x,i}$ and a Block Matrix $BM_{x,i}$ for every group g_x, it is a member of. The updating of $BM_{x,i}$ and $BC_{x,i}$ will follow the conditions *CA1* and *CA2* of section 4.2 restricted to a given group. Besides that, p_i will also maintain a Last Delivered Vector - LDV_i, for each group p_i is a member of. $LDV_{x,i}$ will be updated as specified in the conditions A1, A2, and A3 of section 5.1, and will be used to precisely represent causality between messages sent in a given group (see property *pr2* of section 5.1).

Every message multicast will be timestamped with, in addition to m.b, the current value of the LDV_i vector as m.ldv. The algorithm of the send(m) primitive for a process p_i is given below, in figure 5.3, where g_x represents the destination group of message m, m.g.

```
send(m: non-null-message);
{
  m.s := i;
  m.b := BC_x,i + 1; BC_x,i := m.b;
  m.ldv := LDV_x,i;
  LDV_x,i[i] := BC_x,i;
  BM_x,i[m.b] := '+';
  multicast m to other members of m.g;
}
```

Figure 5.3 - The send primitive

The receive process (figure 5.4) will receive messages from the transport layer and represent them in the appropriate Block Matrix. After that, it signals the deliver process which will try then to deliver the received message. Delivering a message creates conditions for received (and not delivered) messages to be delivered. So, after a message has been delivered, the deliver process is recursively activated until there is no message that can be delivered. A received message m taken from the transport layer and intended for p_i will be immediately represented in the block matrix BM_i related to the group m.g but delivery will only happen when the m.ldv is greater than

or equal to $LDV_{i,x}$ (g_x = m.g). Figures 5.4 and 5.5 show the algorithmic descriptions of the receive and deliver sub-processes, respectively.

```
msg-receive;
{
  do
    receive m from the transport layer;
    update appropriate BM; /* BM_{i,x}[m.b] := m; */
    signal msg-delivery sub-process; /* signal the deliver sub-process */
  od
}
```

Figure 5.4 - The msg-receive sub-process

```
msg-deliver;
{
  if ∃ m in BM_{i,x} | m is not marked delivered and m.ldv ≥ LDV_{i,x} →
    {
      mark m delivered and deliver m to p_i;
      signal msg-deliver sub-process; /* try recursively to deliver another message */
    }
}
```

Figure 5.5 - The msg-deliver sub-process

Maintaining inter-group causal information

Each process p_i maintains a block-counter $BC_{i,x}$ and a block-matrix $BM_{i,x}$ for each group g_x process p_i is a member of. Updating of the Block-Counter and Block-Matrix follows the same rules for the non-overlapping case described in the last paragraph. Thus, $BC_{i,x}$ maintains the largest block-number of messages sent or delivered in group g_x and received/sent messages of g_x are represented in $BM_{i,x}$. Just before a message m is sent in group g_x, it is timestamped with the incremented value of $BC_{i,x}$ as m.b. A destination process, say p_j, on receiving m will know that when m was sent group g_x had progressed in logical time up to block-number m.b. Thus, any message m', $m \rightarrow m'$, subsequently transmitted by p_j to a group g_y, $y \neq x$, will also carry this inter-group causal information so that any process $p_r \in g_x \cap g_y$, $r \neq j$, will deliver m' only after the Causal Block $BM_{r,x}[m.b]$ is complete and its messages delivered. Therefore, enforcing the delivery of m before delivery of m'.

For assessing which Causal Blocks per group should be complete before a received message can be delivered, processes will maintain a vector containing the block-numbers of the largest sent/delivered message per group (the GBN vector), and this information will be transmitted together with any sent message.

The GBN vector

Let $GBN_i[1..k]$ be the process p_i's vector of Group Block Numbers, where k is the number of groups[3]. $GBN_i[j]$ represents the block number of last message sent/delivered in group g_j. Every message sent by process p_i will carry in m.gbn the current value of GBN_i. GBN_i is set up with zeros when p_i starts and will be updated in the following way:

Just after a message m is sent by p_i,

$GBN_i[m.g] := m.b;$

Just after p_i delivers a message m,

$GBN_i[j] := \max\{GBN_i[j], m.gbn[j]\}, j \neq m.g;$

$GBN_i[m.g] := m.b;$

By using the vector GBN, we can now state the message delivery conditions for a given multi-group process p_i. A received message m of g_x can only be delivered if the two following conditions *C01* and *C02* are true.

Conditions for Causal Ordering

C01 : $m.ldv \leq LDV_{x,i}$;

C02 : m.gbn[y] is complete in $BM_{y,i}$, and its messages delivered, for all g_y such that p_i is a member of and $y \neq x$.

By applying the above conditions, protocols satisfying causal order message delivery can be developed.

---

3. In fact, k changes dynamically. Thus, in a real implementation, GBN will be a list of (Gid, BN) pairs, where BN is the block-number of the last sent/delivered message of the group identified by Gid.

Correctness of the protocol

For proving the correctness of the protocol, we have to show that a sent message is eventually delivered (liveness property) and it does so without violating causal order (safety property). As stated before, we assume a reliable FIFO transport layer. Also we assume the time-silence mechanism as in section 4.3.

Safety property

We must show that the delivery of messages by the protocol respect causal order. That is, if two messages m_1 and m_2 are sent, and $m_1 \rightarrow m_2$, m_2 is only delivered in a process p_i after m_1 has been delivered in p_i. *CO1* guarantees that if m1 and m2 belong to the same group gx, m1 is delivered first. Supposing that m1 and m2 are sent in different groups g_x and g_y, respectively, such that $m1 \rightarrow m_2$. *CO2* guarantees that m1 of gx is delivered before m2 of gy is delivered.

CO1 is safe

We will prove *CO1* by contradiction. *CO1* guarantees that a received message m2, $m2.g = g_x$, is only delivered in a process p_j, after $LDV_{j,x} \geq m2.ldv$. Assume that p_j receives m2 and *CO1* is true (i.e. pj delivers m2). Suppose by absurd that there exist a message m1 sent in group g_x, such that, $m_1 \rightarrow m_2$, and m1 has not been delivered by pj (i.e. pj has delivered m2 but not m1). Suppose without loss of generality that m1 was sent by a process pi of g_x, $i \neq j$. Since m2 has been delivered, $LDV_{j,x} \geq m2.ldv$. Thus, by definition (C1 of section 5.1), $LDV_{j,x}[i] \geq m2.ldv[i]$. Also, because $m_1 \rightarrow m_2$, by pr2 of section 5.1, $m1.b \leq m2.ldv[i]$. Therefore, $LDV_{j,x}[i] \geq m1.b$ what implies that m1 must have been delivered (FIFO assumption).

CO2 is safe

We will prove that *CO2* is safe by contradiction. Assume that a message m2, $m2.g = g_y$, has been delivered at a process p_j and suppose by absurd that there exist a message m1, $m1.g = g_x$, $x \neq y$, such that, $m_1 \rightarrow m_2$ and m1 has not been delivered by pj (i.e. pj has delivered m2 of g_y but not m1 of g_x). Note that $GBN_j[x]$ maintains the largest block-number of messages sent or delivered by process p_j in group g_x. Since $m_1 \rightarrow m_2$, $m2.gbn[x] \geq m1.b$. Therefore, when m2 is delivered (by CO2), the Causal

Blocks of $BM_{j,X}[B]$, $B \geq m1.b$ have been completed and all corresponding messages delivered (including m1).

Liveness

To prove the liveness of our protocol, we have to show that a sent message is eventually delivered at the intended destinations. In other words, we must show that once a message m is received, the conditions *CO1* and *CO2* will eventually become true for m. We will prove the liveness of conditions *CO1 and CO2* by induction on all Causal Blocks created in a process p_i.

***CO1* eventually become true :**

base step : suppose the Causal Block number 1 of G_X is created at the group member p_i. Notice that the very first message sent in g_X will be have block number 1. By pr1 of section 5.1, $m.b = \max\{m.ldv[j], 1 \leq j \leq n\} + 1$, where n is the size of m.g. Thus, if $m.b = 1$, $m.ldv[j] = 0$, $1 \leq j \leq n$. Because LDV_i is initialized with zeros, $m.ldv \leq LDV_{i,X}$ is true and m is immediately delivered.

induction step : consider that the Causal Block B of a group g_X is created at a process p_i. That is, a message m is received such that $m.b = B$ and $m.g = g_X$. Suppose by induction hypothesis that all messages m' of g_X with block-number $m'.b < B$ will eventually be delivered at p_i. By pr1 of section 5.1, $m.ldv[j] < B$, $1 \leq j \leq n$. Thus, by our induction hypothesis, the messages of g_X represented in m.ldv will be eventually delivered and therefore $m.ldv \leq LDV_{i,X}$ will eventually become true.

CO2 eventually become true.

Suppose a message m is transmitted in group g_X. By our transport assumption, transmitted messages are eventually received by the destination processes and represented in the appropriate BM. So, assume the m is received by a process p_i of a group g_X. *CO2* will be true when the block m.gbn[y] is complete (and the corresponding messages delivered) in $BM_{i,y}$, for every group g_y that process p_i is a member of. Notice m.gbn[y] is the block-number of a message sent in group g_y. Thus, if p_i belongs to g_y, it will eventually receive a message in g_y with block-number m.gbn[y] (transport assumption). Since the time-silence mechanism guarantees that all

group members will send a message to complete a created Causal Block, then all created blocks will eventually get complete. Therefore, the blocks represented in m.gbn will eventually get complete. We must show now that messages of blocks represented in m.gbn will eventually get delivered. First notice that the very first message transmitted will have m.gbn[y] equal to zero, for any g_y. Thus, the very first message is delivered by liveness of *CO1*. Consider a message m' of a block represented in m.gbn[y], for any g_y. Let us suppose, by induction hypothesis, that messages represented in blocks of m'.gbn will get eventually delivered. That is, *CO2* will eventually become true for m'. Thus, liveness of *CO1* guarantees that m' will eventually be delivered. Therefore *CO2* for m will eventually become true.

5.7 Conclusions

We have presented three approaches for causal order message delivery in overlapping process groups. They represent different trade-offs between delivery delay and message space overhead costs. The Slow causal protocol requires the smallest timestamp (the message block-number) but can potentially introduce large delays since Causal Blocks have to be complete before message delivery. The Fast causal protocol is based on the GLDV vector which is a precise representation of causal dependence between transmitted messages in a multi-group environment. In Fast causal, message delivery occurs as early as possible. That is, the delivery of a message m is only delayed if there is a message m', such that m' $\rightarrow$ m and m' has not been delivery yet. On the other hand, the Fast causal protocol imposes the highest message space overhead among the approaches presented. The Relative causal protocol is a trade-off solution. It provides as early as possible delivery in a uni-group environment but some extra delay is possible in a multi-group environment due to the need of block-completion. When compared with the Fast causal protocol, it provides a smaller timestamp with a potentially slower delivery time. When compared with The Slow causal protocol, the Relative causal provides a faster message delivery but with a higher message space overhead.

Finally, all protocols presented in this chapter have been designed to work in the context of Causal Blocks such that the mechanisms described in another chapters (e.g. time-silence, group-membership, and flow control) can be easily integrated to become part of the causal order protocols.

Chapter 6 - Introducing Fault-Tolerance to Newtop

The Newtop protocol presented in chapter 4.0 delivers messages in each group member in the same total order and always with the same group view. However, when a process crash is considered, it can have the following effects: if the sender crashes during a multicast in a given group, some functioning members may not receive the multicast message; secondly, the Causal Blocks maintained by a process p_i will not complete, when p_i is a member of a group of which the crashed process is also a member. In this chapter, we take Newtop as presented in chapter 4 and extend it in such a way that ordering and liveness is preserved even if the membership changes occur due to (suspected) process crashes. Since the fault-tolerant mechanisms will be developed using the Causal Blocks representation (i.e., using message block-numbers), the extensions we will present can also be applied for the fast causal order protocols described in chapter 5.0, as we will indicate in a subsequent section.

We start in section 6.1 by discussing the requirement of group partitioning for multicast protocols. In section 6.2, we state the fault-tolerant properties of Newtop. In section 6.3, we describe the fault-tolerant mechanisms developed. Section 6.4 discusses related work. Finally, in section 6.5 we conclude the chapter.

6.1 Group Patitioning

As stated earlier, we are interested in a general purpose protocol for asynchronous systems where processes could be geographically widely separated, communicating over a long-haul network such as the internet. As discussed in section 2,1, a fault-tolerant protocol for such an environment, would handle process crashes in the following way: functioning processes *suspect* process crashes and reach agreement only among those processes which they do not suspect to have crashed. (Such protocols are quite different in nature to membership protocols for a synchronous system, e.g., [Cristian91]). Despite efforts to minimise incorrect suspicions by

processes, it is possible for a subgroup of mutually unsuspecting processes to wrongly agree (though rare it may be in practice) on a functioning and connected process as a crashed one, leading to a 'virtual' partition. Thus, there is always a possibility for a group of processes to partition themselves (either due to virtual or real network partitioning) into several subgroups of mutually unsuspecting processes. These observations have motivated us to develop a membership service for Newtop that can support concurrent existence of multiple subgroups, leaving it to applications to decide on the eventual fate of such subgroups.

Most of the existing multicast protocols have not addressed the issue of network partitioning in the manner suggested above. The symmetric protocol of [Amir92b] and the co-ordinator oriented protocol of [Schiper93c] do provide membership services that treat network partitions by permitting the existence of multiple subgroups. In section 6.4 we will compare our membership protocol with these protocols.

6.2 The Fault-Tolerant Properties of Newtop

Let G_i be the set of groups P_i belongs to: $G_i = \{g_x \mid P_i \in g_x\}$. When P_i multicasts (or delivers) a message m with $m.g = g_x$, it actually does so only to (or from) those processes which it *views* as functioning members of g_x. When g_x is initially formed, each functioning P_i installs an identical, initial view $V^0_{x,i} = \{P_1, P_2, \ldots P_n\}$. As P_i 'justifiably' *suspects* another P_k to have crashed or disconnected, it attempts to confirm its suspicion with other members it does not suspect; if confirmed, it installs a new view that does not include P_k. Let $V^0_{x,i}$, $V^1_{x,i}$, $V^2_{x,i}$, ... $V^r_{x,i}$ be the series of views P_i has thus sequentially installed over a period of time, until it crashes. (The view does not exist for a crashed process.) The view installations by P_i must satisfy certain conditions so that message delivery by Newtop can be 'atomic' with respect to view updates. For this purpose, Newtop provides each P_i with a *group-view* process, denoted as $GV_{x,i}$, for each g_x, $g_x \in G_i$. Group-view processes execute a *membership protocol* to reach agreement and update group views.

The Newtop membership protocol maintains consistency in the presence of (real or virtual) partitions by permitting a group of processes to partition themselves into two or more sub-groups of connected processes with the property that: (i) the functioning processes within any given subgroup will have mutually consistent views about the membership; and (ii) the views of processes belonging to different subgroups are guaranteed to stabilise into non-intersecting ones. Newtop leaves it to applications to decide whether or not the applications should continue to maintain more than one subgroup. This flexibility makes Newtop more powerful than many other protocols [Melliar-smith91, Ricciardi91, Mishra91] that can guarantee continued group operation only when the group partitions in such a way that exactly one subgroup can be uniquely identified as the primary. This in turn requires a certain number (at least a majority) of processes in the group to remain operational and connected; as we have discussed earlier, this requirement is not always possible to meet.

In Newtop, view updates performed by processes of a group g_x satisfy the following *view consistency* properties:

VC1: The sequence of views installed by any two functioning processes that do not suspect each other are identical *(validity)*.

VC2: If P_k, $P_k \in V^r_{x,i}$, crashes or gets disconnected from P_i and if P_i does not crash, then P_i will eventually install $V^{r'}_{x,i}$ such that $r' > r$ and $P_k \notin V^{r'}_{x,i}$ *(liveness)*.

VC3: any two functioning processes deliver the same set of messages between two consecutive views that are identical. That is, $V^{r-1}_{x,i} = V^{r-1}_{x,j}$ and $V^r_{x,i} = V^r_{x,j} \Rightarrow$ the set of m, $m.g = g_x$, delivered by P_i and P_j in V^{r-1}_x are identical. This condition is the generalised version of *virtual synchrony* defined for the ISIS system [Birman91b]; the generalisation is explained below.

Consider P_i multicasting *m* in view $V^r_{x,i}$. Let this event be denoted as $send_i(m,r)$. P_i delivers its own messages only by executing the protocol. Suppose that P_i delivers *m* in view $V^{r'}_{x,i}$, for some $r' \geq r$; denote this event as $delivery_i(m,r')$. Note that $r' \geq r$, and the '$\rightarrow$' relation (defined only on *send* and *delivery* events) is not closed under view updates. The *virtual synchrony* model of ISIS requires $r' = r$. In section 6.3, we show that our protocol can be modified to provide this closure property, but only at the necessary expense of performance, by blocking *send* operations when a new view is being installed (this is the case in ISIS as well). Newtop has the following message delivery properties (in stating them the suffix x has been dropped):

validity: *for any m:* $send_i(m,r) \wedge delivery_j(m,r') \Rightarrow P_i \in V^{r'}_j$. In words: a process will deliver a message *m* in view V, only if the sender of *m* is in V.

liveness: if a P_i sends *m* in view V^r_i, then provided it continues to function, it will eventually deliver *m* in some view $V^{r'}_i$, $r' \geq r$.

atomicity: $\forall P_i, P_j$ *s.t.* $V^r_i = V^r_j \wedge V^{r+1}_i = V^{r+1}_j : delivery_i(m,r) \Leftrightarrow delivery_j(m,r)$. This property is implied by VC3.

The three properties stated above (validity, liveness, and atomicity) guarantee live, atomic delivery in the presence of dynamic membership changes. The total order delivery property stated in section 4.2 has now to be redefined to take into account membership view changes. The order property stated below ensures total order deliveries in single groups. Subsequently, we extend this property for multiple process groups.

Total order delivery (single group) : $\forall P_i, P_j$ *s.t.* $V^r_i = V^r_j \wedge V^{r+1}_i = V^{r+1}_j$: $delivery_i(m,r) \rightarrow delivery_i(m',r) \Leftrightarrow delivery_j(m,r) \rightarrow delivery_j(m',r)$ and $m \rightarrow m' \Rightarrow delivery_i(m,r) \rightarrow delivery_i(m',r)$.

Newtop extends the above property so that processes simultaneously belonging to multiple groups are delivered messages in total order. Let m' be a message with m'.g = g_y and $p \geq 0$ be an integer:

Total order delivery (multiple groups) : $\forall P_i, P_j$ *s.t.* $V^r_{x,i} = V^r_{x,j} \wedge V^{r+1}_{x,i} = V^{r+1}_{x,j} \wedge V^p_{y,i} = V^p_{y,j} \wedge V^{p+1}_{y,i} \wedge V^{p+1}_{y,j}$: $delivery_i(m,r) \rightarrow delivery_i(m',p) \Leftrightarrow delivery_j(m,r) \rightarrow delivery_j(m',p)$, and $m \rightarrow m' \Rightarrow delivery_i(m,r) \rightarrow delivery_i(m',p)$.

6.3 Making Newtop Fault-Tolerant

We now describe how to extend Newtop (as in chapter 4) to incorporate fault tolerance: ordering and liveness is preserved even if membership changes occur due to (suspected) process failures. As stated before, Newtop provides each P_i with a *group-view* process, denoted as $GV_{x,i}$, for each g_x, $g_x \in G_i$. $GV_{x,i}$ is responsible for maintaining P_i's view of the group membership of g_x. Informally, this extension has the following aspects: (i) the liveness mechanism based on timesilence (section 4.3) needs to be extended to enable $GV_{x,i}$ to *suspect* a failure of some remote process (P_j) that does not seem to be responding; (ii) in which case $GV_{x,i}$ can initiate a *membership agreement* on P_j, the outcome of which is that either processes agree to eliminate P_j from the group view, with an agreement on the last message sent by P_j, or Pj continues to be a member and P_i is able to retrieve missing messages of P_j; (iii) the time-silence mechanism described in section 4.3 will be extended to keep processes lively in sending messages during the execution of the membership protocol (during the periods where there are no application or membership related messages to be transmitted). The extension for the time-silence mechanism is as follows: the time-silence mechanism related to a process P_i will force P_i to multicast a null message if no (null or non-null) message was sent by P_i in the past interval of length, say τ. This

extension is necessary in order to guarantee that new crashes are 'promptly' suspected while running the membership protocol.

6.3.1 Message Stability

It is necessary to ensure that a process can always retrieve a missing message from another functioning member process. This in turn means that we require a mechanism that enables a process to safely discard a received message. To develop such a mechanism, we will first define the concept of *message stability*:

Message Stability: A Causal Block $BM_{x,i}[\beta]$ becomes stable if it is known to be complete in all the processes in the current view of g_x. The messages represented in $BM_{x,i}[\beta]$ will also be termed stable.

Blocks become stable in increasing order of block-numbers: if block $BM_{x,i}[\beta]$ is stable, then all blocks $BM_{x,i}[\beta_0]$, $\beta_0 < \beta$, will also be stable. Once a Causal Block becomes stable at P_i, P_i "knows" that the corresponding messages have been received by all the members in P_i's view. Stability information is passed together with transmitted messages. That is, when a message m, $m.g = g_x$, is transmitted by P_i, a field *m.lcb* is used to represent the number of the *largest complete block* in $BM_{x,i}$. To compute stable blocks, each process P_i, maintains a vector called $SV_{x,i}$ (Stability Vector) for each g_x. At process P_i, $SV_{x,i}[j]$ represents the largest complete block at P_j. If $\min(SV_{x,i})$ represents the minimum value in $SV_{x,i}$, then all blocks $BM_{x,i}[\beta]$, $\beta \leq \min(SV_{x,i})$ will be stable. For the sake of fault tolerance, a block that is not stable, and the messages represented in it are not discarded from the local storage of processes.

6.3.2 Managing Group Membership

The membership algorithm of Newtop is based on the approach used in Psync [Mishra91, Mishra93], adapted to the context of Causal Blocks and extended to coordinate view updates with message delivery. Group-view process $GV_{x,i}$ of P_i works as if P_i is not a member of any other group; it refers only to the local Block

Matrix corresponding to g_x. So, we can ignore the fact that P_i can be a member of more than one group, and will describe the $GV_{x,i}$ of P_i for a given g_x, dropping the suffix x.

Each GV_i has a failure *suspector* module, S_i, which is implemented as follows: soon after a new causal block is created in BM_i, S_i sets a timeout for duration ω, $\omega >$ *timesilence* timeout (local-time-silence - section 4.3). If the block is not complete within ω, S_i suspects those member processes whose messages are needed for the completion of that block as crashed and notifies GV_i of its suspicion. In practice, ω should be tuned to a value that minimises the possibility of unfounded suspicions.

We will assume that a send primitive *mcast(m)* is available to P_i for multicasting *m* for total order message delivery (ie., for P_i to execute a $send_i(m)$ event). GV_i also uses this *mcast(m)* primitive to multicast a membership related message. However, at the receivers, these membership messages are treated differently. Each P_i has a sub-process, *msg-receive*, that is responsible for receiving messages from the transport layer and maintaining the Block Matrix. The *msg-receive* sub-process treats the membership messages destined for the local GV process as null messages for the purposes of representation in the Block Matrix; it passes the actual messages directly to the local GV process.

A notification from S_i to GV_i will be of the form $\{P_k, lbn\}$ - indicating that P_k is suspected to have crashed and lbn is the block-number of the last message P_i has received from P_k. GV_i maintains a set $suspicions_i$ where notifications from S_i are entered. GV_i also multicasts a *suspect* message $(i, suspect, \{P_k, lbn\})$ to GV processes of all processes (including GV_k) that are in its current membership view V_i. If GV_i receives confirmation that all other unsuspected members in V_i also suspect each $\{P_k, lbn\}$ in its $suspicions_i$, it decides to treat each P_k of $suspicions_i$ as having failed and P_k is added to a set called $failed_i$. The msg-receive sub-process of P_i discards any messages received from P_k and GV_k, if either $P_k \in failed_i$ or $P_k \notin V_i$. Also, once suspicion $\{P_k, lbn\}$ has been added to $suspicions_i$, it will keep the messages received from P_k and GV_k as pending. If suspicion $\{P_k, lbn\}$ is subsequently

refuted, the pending messages will be assumed to have been just received, and will be handled appropriately; if, however, suspicion $\{P_k, lbn\}$ is confirmed as a failure, then the pending messages of P_k and GV_k are discarded.

Suppose that GV_j receives the message *(i, suspect, {P_k, lbn})* from GV_i. If $\{P_k, lbn\}$ is already in $suspicions_j$, GV_j regards GV_i as yet another process that holds the same suspicion as itself; if however $\{P_k, lbn\}$ is not in $suspicions_j$, it records this suspicion from P_i, but suspends judgement on it pending confirmation from its own S_j. If in the mean time P_j receives a message m from P_k with $m.b > lbn$, then GV_j multicasts a *refute* message *(j, refute, {P_k, lbn})*. When GV_i receives this refute message, it stops suspecting P_k for lbn, and removes $\{P_k, lbn\}$ from $suspicions_i$; it also initiates an attempt to recover the missing messages of P_k (a missing m can be piggybacked in the refute message; by definition any missing m is unstable, so would not have been discarded by P_j; P_j can therefore always piggyback m.); after recovery, P_i multicasts *(i, refute, {P_k, lbn})* message. If GV_i ever receives a message *(k, suspect, {P_i, lbn_i})*, it takes no action in the hope that some GV_j will refute that suspicion.

The event driven algorithm for GV_i is given below, dropping the suffix i for all the set variables used exclusively by GV_i; these set variables are initialised to empty and a boolean variable *consensus* is initialised to *false*, when the group g_x is formed. The algorithm describes the steps taken by GV_i, once a certain condition holds.

(i) *notification {P_k, lbn} received from S_i*: suspicions := suspicions $\cup$ $\{P_k, lbn\}$; mcast(i, suspect, $\{P_k, lbn\}$);

(ii) *(j, suspect, {P_k, lbn}) received*: **if** ($P_k \neq P_i$) **then** record the suspicion $\{P_k, lbn\}$ of GV_j; **if** ($P_k = P_i$) **then** discard the received message;

(iii) *suspicion {P_k, lbn} of GV_j is recorded* $\wedge$ *m, m.b > lbn, from P_k is represented in BM_i*: mcast(i, refute, $\{P_k, lbn\}$); /* all received m of P_k, $m.b$>lbn, are piggybacked */

(iv) *(j, refute, {P_k, lbn}) received* $\wedge$ *{P_k, lbn}* $\in$ *suspicions*: suspicions := suspicions – $\{P_k, lbn\}$; recover the missing m, $m.b$>lbn of P_k; mcast(i, refute, $\{P_k, lbn\}$);

(v) *for every* $\{P_k, lbn\} \in suspicions$, if there exist *suspect messages received from every* GV_j *of* $P_j \in V_i - (\{P_k \mid \{P_k, lbn\} \in suspicions\} \cup failed)$: detection := suspicions; suspicions := { }; mcast(i, confirmed, detection); consensus := true;

(vi) $(j, confirmed, detection_j)$ *received* $\wedge$ $detection_j \subseteq suspicions$: detection := $detection_j$; suspicions := suspicions - $detection_j$; mcast(i, confirmed, detection); consensus := true;

(vii) *(consensus = true):* lbn_{mn} := minimum{lbn | $\{P_k$, lbn$\} \in$ detection}; for every $P_k \in \{P_k$, lbn$\} \in$ detection **do** instruct *msg-receive* sub-process to assume that it has received and will receive from P_k only a null m, m.lcb := ∞, for every m.b > lbn_{mn}; failed := failed $\cup$ $\{P_k\}$; removals := removals $\cup$ $\{P_k, lbn_{mn}\}$ **od**; instruct P_i to mcast(remove, failed); consensus := false;

(viii) $(j, confirmed, detection_j)$ *received* $\wedge$ $(P_i, lbn\} \in detection_j$ *for some lbn*: force S to suspect P_j for lbn_j = block-number of the received message; suspicions := suspicions $\cup$ $\{P_j, lbn_j\}$; mcast(i, suspect, $\{P_j, lbn_j\}$) ensuring that this multicast has a block-number larger than lbn_j;

(ix) P_i *delivers* $m = (remove, failed)$: F := *failed* of delivered *m*; **if** $F \cap V_i \neq \emptyset$ **then** for each $P_k \in F \cap V_i$ **do** remove the column in BM corresponding to P_k, starting from the row number lbn_k, such that $\{P_k, lbn_k\} \in$ removals; removals := removals - $\{P_k, lbn_k\}$; failed:= failed - $\{P_k\}$ **od**; $V_i := V_i - (F \cap V_i)$;

When GV_i confirms all of its suspicions (condition (v)) or a subset of them (condition (vi)) into agreed failure detection, it sets the boolean *consensus* to true. Functioning members that hold identical views and do not suspect each other, will confirm identical *detection* sets in an identical order. (For the proof of this, see [Mishra91].) Whenever a new *detection* set is agreed, GV_i instructs (in (vii)) *msg-receive* sub-process to assume that it can avail from every detected process P_k a stream of consecutively block-numbered, null messages starting with the block-number $lbn_{mn}+1$, where lbn_{mn} is the minimum of $\{lbn \mid \{P_k, lbn\} \in detection\}$. These 'virtual' messages will have their *lcb* field set to ∞, and will serve the purpose of completing any blocks for which messages from P_k are necessary. So, this instruction by GV_i to the *msg-receive* sub-process causes block completion and stabilisation to progress and the message delivery to P_i to resume, after being temporarily halted by

the absence of messages from suspected members. Further in (vii), GV_i includes P_k into the set $failed_i$ and $\{P_k, lbn_{mn}\}$ into the set $removals_i$ to remove (later) failed P_k from its view assuming that no (actual) m was received from P_k for any $m.b > lbn_{mn}$. This assumption is necessary to ensure safety (an example is given later to illustrate this point).

The *remove* message R_i = *(remove, failed$_i$)* multicast by P_i is delivered in identical order to all functioning and connected members who have agreed over the set $detection_i$. Note that this *remove* message R_i will have a block-number larger than lbn_{mn}. This is because S_i notifies suspicions by identifying member processes whose messages are needed for the completion of a given block in BM_i, after having waited for ω time units after that block was first formed. So, for every $\{P_k, lbn_k\}$ notified by S_i to GV_i, P_k has been found responsible for the non-completion of a block $BM_i[B]$, $B > lbn_k$. As $\omega >$ the timeout for $timesilence_i$, when S_i notified $\{P_k, lbn_k\}$, P_i must have sent a null or non-null m, $m.b > lbn_k$. Further, when S_i is forced to suspect $\{P_k, lbn_k\}$ in (viii), GV_i ensures that the *suspect* message it multicasts has a block-number larger than lbn_k. (This is possible as BC_i can be advanced by any non-zero, positive integer before block-numbering a message to be multicast.) Thus, R_i will always have a block-number larger than lbn_{mn} computed just before it is multicast. This means that when R_i = *(remove, failed$_i$)* is delivered to any $P_j \in V_i$ - $failed_i$ of R_i, P_j will have no non-null m, $m.b \geq R_i.b$, from $P_k \in failed_i$ of R_i, that is waiting to be delivered. (Any non-null m, $m.b \geq R_i.b$, received from P_k, would have been turned into a virtual null message.) So, upon delivering R_i, if P_j prompts GV_j to update its view by removing all $P_k \in failed_i$ of R_i, the validity property (of section 6.3) is guaranteed to be satisfied.

For safety reasons, GV_i treats all the processes in a given set $detection_i$ as having failed "together", ignoring any outputs produced by them with $m.b > lbn_{mn}$. This is necessary to preserve causal order delivery, since it is possible that one such output could have been produced by a process after consuming an output of another failed process that no one else received. The following example illustrates this:

Suppose that functioning P_i and P_j hold identical views and never permanently suspect each other. Let P_r crash during the multicast of m, and only P_s receives m. Let P_s deliver m (possible, if the arrival of m from P_r causes $m.b$ block to become complete), multicast m' that is received by P_i and P_j, and crash before it could refute the suspicion $\{P_r, lbn\}$ for some $lbn < m.b$, held by GV_i and GV_j. P_r and P_s will be detected together by GV_i and GV_j, and m' from P_s will be replaced by a virtual null message, as $m'.b > m.b \geq lbn_{mn}$. Thus, m', $m \rightarrow m'$, is guaranteed not to be delivered when m cannot be delivered.

6.3.3 Example of an Execution of the Group Membership Algorithm

Now we present an example to illustrate how the membership algorithm works. Consider a group $g = \{ P_i, P_j, P_k, P_l, P_m\}$ in which each functioning member holds the initial view $V^0 = g$. Say, P_i and P_j never suspect each other and also, P_k and P_l never suspect each other. Let P_i, P_j, P_k and P_l suspect P_m for some lbn_m. Let the suspect messages of GV_i and GV_j be received by GV_k and GV_l, before a network failure partitions P_i and P_j from P_k and P_l. GV_i and GV_j, not receiving any suspect message for $\{P_m, lbn_m\}$ from GV_k and GV_l, suspect $\{P_k, lbn_k\}$ and $\{P_l, lbn_l\}$, and form $detection_i = detection_j = \{\{P_m, lbn_m\}, \{P_k, lbn_k\}, \{P_l, lbn_l\}\} = detection_{ij}$ (say), with $failed_i = failed_j = \{P_m, P_k, P_l\} = failed_{ij}$ (say) (see figure 6.1 - s_r represents *suspect* $\{p_r, lbn_r\}$).

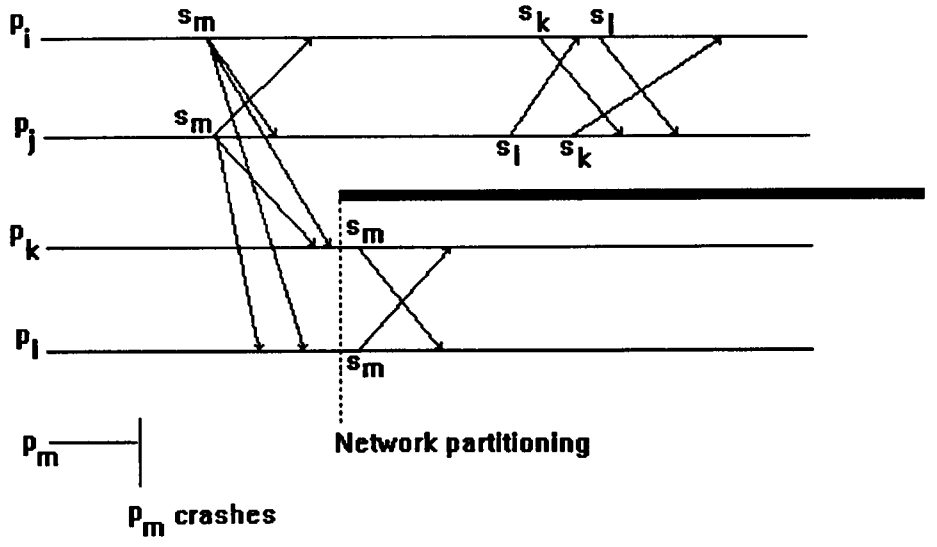


Figure 6.1 - Example of Network Partitioning

GV_k and GV_l, having received a suspect message for $\{P_m, lbn_m\}$ from GV_k and GV_l, do not (yet) suspect P_i and P_j, and proceed to form $detection_k = detection_l = \{\{P_m, lbn_m\}\} = detection_{kl}$ (say), with $failed_k = failed_l = \{P_m\} = failed_{kl}$ (say). Both P_i and P_j will multicast *(remove, $failed_{ij}$)* and both these messages will be delivered to P_i and P_j in an identical order. Let R_{ij} be the *(remove, $failed_{ij}$)* message that is first to be delivered to P_i and P_j. Similarly, let R_{kl} be the *(remove, $failed_{kl}$)* message that is first (of two) to be delivered to P_k and P_l. By delivering R_{ij}, P_i and P_j will update their views to $V^1 = \{P_i, P_j\}$, and before the update, they would have delivered an identical set of non-null m, $m.b \leq R_{ij}.b$; the messages from P_m, P_k and P_l in the block-number range $min\{lbn_m, lbn_k, lbn_l\} < m.b \leq R_{ij}.b$ will only be virtual null messages. Similarly, P_k and P_l would have delivered an identical set of non-null m, $m.b \leq R_{kl}.b$, when they updated their views to $V^1 = \{P_i, P_j, P_k, P_l\}$.

Note that V_i^1 and V_k^1 are not mutually exclusive as the network partition was not observed in the same order with respect to the observation of P_m's failure: concurrently by P_i and sequentially by P_k. This situation is short lived as GV_k and GV_l must subsequently suspect P_i and P_j: either by receiving *(i or j, confirmed, $\{P_m, P_k, P_l\}$)* and executing (viii) if the network partition is transient, or by being notified

from the local suspector if the network partition is permanent. P_k and P_l will reach agreement on *detection'k* = *detection'l* = {{P_i, lbn$_i$}, {P_j, lbn$_j$}} = *detection'kl* (say). The delivery of *R'kl* = *(remove, {Pi, Pj})* will update their views to V^2 = {P_k, P_l}; VC2 is met, as P_k and P_l have disconnected P_i and P_j in their V^2. This also shows that after a transient period of instability caused by successive failures, partitioned subgroups will stabilise their views to non-intersecting ones. In view V^1, P_k and P_l would have delivered an identical set of non-null *m*, $R_{kl}.b \leq m.b \leq R'_{kl}.b$, that were sent only by processes in V^1. Thus VC3 is met. That VC1 is met can be seen by the fact that identical view changes are installed by P_i and P_j, and also by P_k and P_l.

6.4 Comparison with Existing Related Work

In this section, we highlight some significant differences between our protocol and the published ones that support concurrent existence of multiple, partitioned subgroups. (We will refer to the previous example for this purpose.) The protocol of [Schiper93c] has the following two aspects: (i) concurrent views are non-intersecting (i.e., in the example, the situation where V^1 of P_k and P_l intersecting with V^1 of P_i and P_j will not occur); and (ii) message deliveries respect the original (restricted) virtual synchrony property (i.e., no message sent in view V^r is ever delivered after V^{r+1} is installed). As we show below, our protocol can be modified easily to guarantee these properties; further, the original virtual synchrony is met by imposing less severe performance penalty than [Schiper93c].

Concurrent non-intersecting views are guaranteed in [Schiper93c] essentially by defining a process view as a set of process *signatures*, where a signature is a tuple: {process-id, sequence-number}. Let GV_i replace V_i by ϑ_i = {{P_j, e_i} | $\forall P_j \in V_i$}, where e_i is the total number of processes GV_i has excluded from the initial view; e_i=e_j, if V_i^r=V_j^r for every $r \geq 0$. Thus, in the example, ϑ^0={{P_i, 0}, {P_j, 0}, {P_k, 0},{P_l, 0},{P_m, 0}} for all functioning processes of *g*. After partitioning, ϑ_i^1= ϑ_j^1={{P_i, 3}, {P_j, 3}} which do not intersect with ϑ_k^1=ϑ_l^1={{P_i, 1}, {P_j, 1}, {P_k, 1},{P_l, 1}}; after stabilising, ϑ_k^2=ϑ_l^2={{P_k,3},{P_l,3}}.

To meet the original virtual synchrony property, two modifications are needed. First, view updates should be attempted only on the *last* delivery of a given *(remove, failed)* message; other earlier deliveries will prompt GV_i to take no action except to keep the count. (Currently, view update is attempted upon the delivery of any *(remove, failed)* message.) Referring to the example, the message R_{ij} whose delivery should prompt a view update for P_i and P_j will be: the π^{th} *(remove, failed$_{ij}$)* message to be delivered by P_i and P_j where π is the size of the new view proposed by *(remove, failed$_{ij}$)* and π = | *current-view* - *failed$_{ij}$* | = 2. So, when P_i and P_j deliver the second of the two *(remove, failed$_{ij}$)* messages, they will update their views to $V^1 = \{P_i, P_j\}$. Similarly, the message R_{kl} whose delivery should prompt a view update for P_k and P_l will become the fourth *(remove, failed$_{kl}$)* message to be delivered, as *failed$_{kl}$* = $\{P_m\}$ and $|V^0$ - *failed$_{kl}$* | = 4. (Note that P_i and P_j will not multicast *(remove, failed$_{kl}$)* and hence R_{kl} does not exist for P_k and P_l to update their views to V^0 - *failed$_{kl}$* . This does not lead to deadlock as GV_k and GV_l will subsequently suspect P_i and P_j, agree on *detection'$_{kl}$* = $\{\{P_i, lbn_i\}, \{P_j, lbn_j\}\}$, and instruct P_k and P_l (respectively) to multicast *(remove, failed'$_{kl}$)* where *failed'$_{kl}$* is now $\{P_i, P_j, P_m\}$. The second delivery of *(remove, failed'$_{kl}$)* will cause P_k and P_l to update their views to $V^1 = V^0$ - *failed'$_{kl}$* = $\{P_k, P_l\}$).

The second modification is that P_i should be blocked from multicasting any non-null messages (except *remove* messages) during the period when the set variable *failed$_i$* remains non-empty. So, P_i may well be blocking (application related) multicasts for a (possibly) long period, when its successive detections cascade with each other before resulting in a view update (as the detections $\{\{P_m, lbn_m\}\}$ and $\{\{P_i, lbn_i\},\{P_j, lbn_j\}\}$ of P_k and P_l cascade into *failed'$_{kl}$*). This performance penalty is less severe compared to [Schiper93c] where P_i is required to halt multicasting whenever either *suspicions$_i$* or *failed$_i$* becomes non-empty.

Newtop can knowingly discard certain messages received from a detected P_k, as not having received (every non-null m from P_k with $lbn_{mn} \leq m.b \leq lbn_k$, $\{P_k, lbn_k\} \in$ *detection$_i$*, will be discarded by P_i). This is because the multicasts do not carry any

explicit information about causally preceding messages, and received messages are represented in a matrix. In Transis, messages carry causal precedence information and are represented in a directed acyclic graph (DAG) that precisely indicates the presence or absence of causal relation between messages. Using DAG, Transis [Amir92a] provides an efficient causal delivery service which is not the primary objective of Newtop. Our matrix representation, while considerably simplifying the protocol implementation, does not completely reveal the *absence* of causal relations: two concurrent messages represented in different blocks, are treated as causally related (of course, this does not matter in a total order protocol). So the protocol of [Amir92b] is more precise than ours in rejecting messages received from processes that are to be removed from the view. Finally, the protocols of [Schiper93c, Amir92b] deal with process recoveries and merging of partitioned subgroups, which we have not considered here.

6.5 Conclusions

We have presented an approach based on the concept of Causal Blocks for implementing a fault-tolerant total order multicast protocol. We have shown how message order can be preserved in an asynchronous environment despite the occurrence of process crashes and network partitions. The membership protocol ensures that network partitions do not lead to processes forming inconsistent group membership views; further, message delivery is kept atomic with respect to view change installations. We have highlighted the significant differences between our membership protocol and the similar protocols published in the literature.

The fault-tolerant mechanisms have been developed using the Causal Block representation (i.e., using message block numbers). The causal order protocols described in chapter 5 can, therefore, incorporate fault-tolerance by running Suspector and Group View processes as described for Newtop. In the absence of failures, messages will be delivered using the causal order protocol (either Slow, Fast, or Relative causal order protocol). When a process failure is suspected, the

membership protocol (Group View processes) is switched on and message delivery is suspended until a (possibly) new membership is established. After that, message delivery can resume using the causal order protocol.

Chapter 7 - The Implementation of Newtop

Newtop has been implemented and tested over a set of networked UNIX workstations (sun sparc stations). The system is divided into two layers: Newtop and the multicast transport layers. A message to be sent by the Newtop protocol is passed first to the multicast transport layer that then transmits it reliably[1]. Accordingly, Newtop gets messages intended to application processes through the reliable multicast layer. The architecture of the Newtop consists basically of concurrent processes communicating through messages-queue UNIX IPCs. Before showing the implementation of the concurrent processes that make the Newtop protocol, we describe how the transport multicast layer has been implemented.

7.1 The Transport Multicast Layer

A message, once sent by the transport multicast layer, will be received by all functioning destinations as long as the sender of the message does not crash. That is, if the message sender crashes before finishing the message multicast, some of the destination processes may not get the message. An initial version of the transport multicast layer has been implemented by creating multiple point-to-point TCP/IP sockets[2] and daemon processes to send/receive multicasts in a given group. The identifiers of sockets for a given group are generated by applying a function to the unique identifiers of the group processes. A message to be transmitted by the transport layer is passed to the *sending message queue*[3]. Then, the multicast sender (a daemon process), which is installed before the execution of the Newtop, takes the message from the *sending message queue* and sends it to all group members except

[1]As long as the sender of a message does not crash, the sent message is received by all functioning destinations and in FIFO order.
[2]TCP/IP channels (UNIX stream sockets) are reliable and deliver messages in FIFO order [Stevens90].
[3] It could be a UNIX pipe, a file, or even a shared variable. Message queues as well a C++ class to implement them are described later in this text.

the application process that originated the message. Each application process has got a multicast receiver (a daemon process) associated to each of the group members. Once a message is received by a multicast receiver, it is passed to the *receiving message queue* and then consumed by Newtop. Hence, Newtop communicates with the transport layer through two message queues, the *sending* and the *receiving* message queues. The identifiers of these message queues related to a given group are known by the Newtop layer and are derived from the application process unique identifier supplied on the creation of the group. We present below the simplified code of the sender and receiver multicast transport layer processes.

```
// The sender process
main() {
    // this is the installing  procedure for the sender process
    multicastList GL;          // GL will  contain the description of the group members
    GL.num_participants = groupSize.
    GL.multicast_UID = "multicastUID";
    GLmembers[0] = UID1;
    GL.host[0] = "host of process UID1";
    .....                      // initialize other attributes of GL ...
    message_queue sendMulticast("myUID" + 101);   // creates the sending message queue
    tcpclient client[MaxNumPart];   // create empty client socket structures
    for (int i =0; i < groupSize; i++)
    {
      if (i !="index for myUID")
        //create a sending socket for the channel "myUID"  <--> GL.members[i].
        client[i].init(  (int) GL.members[i] + (200 + "index for myUID"), GL.host[i]);
    }

    int childPid = fork();      // forks a new process (the sender process)
    if (childPId == 0) // this is the multicast sender process
      while(1) // do it forever
        {
          n = sendMulticast.receive(buf);
          if (i !="index for myUID")
            client[i].send( &buf);
        }
} // end of the simplified code for the sending process

// The receiver process
main() {
   // this is the installing procedure to the receiver process
    multicastList GL;
    GL.num_participants = groupSize.
    GL.host[0] = "host of process UID1";
    .....    // initialize other attributes of GL ...
    message_queue receiveMulticast("myUID" + 100);   // creates the sending message queue
    tcpserver server[MaxNumPart];   // create empty server socket structures
   for (int i =0; i < groupSize; i++)
    {
      if (i !="index for myUID")
        server[i].init(  (int) "myUID"  + (200 + i) );
    }

    int childPid = fork();      // forks a process
    if (childPId == 0) // this is the multicast receiver process
      while(1) // do it forever
        {
          int size = server[i].recv( &buf);
          receiveMulticast.send(&buf, size);
        }

} // end of the simplified code for receiver process
```

Rules for creating application process unique identifies

Since the transport multicast layer creates structures (and the corresponding unique identifiers) such as messages-queues and UNIX sockets to work on specific group processes, there must be a way of mapping those structures to the corresponding group processes. This is done by deriving structure identifiers from the unique application group process identifiers. This, however, imposes some restrictions on the way application processes identifiers can be generated. Let UID_i be the process P_i unique identifier. Thus, for any application processes P_i and P_j, $i \neq j$, $UID_i - UID_j \geq K$, where K must be large enough so as to accommodate all identifiers of structures (message queues and sockets) used by the multicast and Newtop layers. In the current implementation, K has been set to 1000. Following is the distribution of identifiers derived from the application process unique identifier. Assume the existence of an application process unique identifier, UID. Derived structures unique identifies are generated as follows:

UID + [1..100) are IPC message queues;

UID + [100, 101] are multicast message queue identifiers (send and receive queues); and

UID + [200..K) are identifiers of sockets used to implement the reliable multicast.

As an example, assume a group of 3 processes, P_0, P_1, and P_2. Also, assume that the unique identifiers of these three processes are 10000, 20000, and 30000 respectively. Then, the following identifiers will be generated in the transport layer.

receive multicast queue

P_0 : 10100; P_1: 20100; P_2: 30100

send multicast queue

P_0 : 10101; P_1 : 20101; P_2 : 30101

client sockets (send multicasts)

P_0 : 20200 for P_1 and 30200 for P_2.

P_1 : 10201 for P_0 and 30201 for P_2.

P_2 : 10202 for P_0 and 20202 for P_1.

server sockets (receive multicasts)

P_0 : 10201 for P_1 and 10202 for P_2.

P_1 : 20200 for P_0 and 20202 for P_2.

P_2 : 30200 for P_0 and 30201 for P_1.

7.2 The Newtop Layer

Newtop operates through a set of concurrent processes that communicate to each other through message-queue UNIX IPCs. A message intended to processes using Newtop, is received by the receiver process which gets the message from the *receiving message-queue* provided by the transport layer. After receiving a message, the receiver process pass it to the deliver process. Also the receiver process passes the header of the message (message block-number, group indentifier, etc.) to the following processes : local-time-silence, suspector, transmitter, and membership. When the message is ready to be delivered (see chapter 4), it is passed from the deliver process to the application process through the *application queue*. The application process, then consumes the message.

A message to be sent by Newtop, is passed to the transmitter process through the *transmitter queue*. After block numbering the message, preparing other fields of the message header, and checking for flow control conditions (see chapter 8), the transmitter process, then, pass the message to the transport multicast layer (by putting it into the *transmitting message-queue*) to be finally transmitted. After that, the transmitter process pass the header of the message to the processes: deliver, local-time-silence, and suspector. Before describing the processes, we describe the main aspects of the relevant C++ classes and data structures used throughout the implementation.

7.2.1 Message Queues

Message queues are IPC mechanisms of UNIX System that appeared for the first time in the System V. These queues are created and manipulated by system calls and

reside in the system kernel [Stevens90]. The C++ class message_queue implements the IPC message queue operations, such as creation of a queue, removal of a queue, sending message to a queue, receiving message from a queue, etc. Below is the simplified class definition code of message_queue.

```
class message_queue {
 public:
   message_queue(ket_t key);     // create a queue unique identified by key (32 bits).
   ~message_queue();              // destructor
   send(char *ptr, int size);     // send a message
   receive(char *ptr);            // receive a message
};
```

7.2.2 The Block Matrix

Messages belonging to unstable blocks are kept in a pool called BM (Block Matrix) and physically organizes in a two level data structure (figure 7.1). The first level of BM is a hashing table addressed by block-numbers. Each entry of the hashing table contains a Causal Block number and the pointers to the messages associated with that Causal Block. These messages are kept in a dynamically allocated message pool (the second level). The size of the hashing table, say N, is known when the group is created[4]. When a non-null message m is to be entered in BM (or retrieved from), the hashing function h, where h(m.b) = m.b mod N, will produce the address in the hashing table where the pointers to the messages of block m.b can be found. Collision of addresses (block b and block b + N, for instance) never happens due to the flow control provided (see chapter 8). The functionality of BM is implemented through a C++ class called Block_Matrix with the following member functions : insert(*ptr), stabilize(int bn), first_message(int bn), and next_message(bn). The first function puts a new message in BM, the second stabilize a Causal Block (i.e. discards their messages from the message poll and the corresponding entry from the hashing table). The last two functions are used to retrieve messages to be delivered from complete blocks. The following is the simplified code for the class Block_Matrix. definition.

[4] This is defined in the configuration file : "define.h".

```
class Block_Matrix {
.... some data structures
public:
  Block_Matrix();
  ~Block_Matrix();          // destructor
  void insert(message_buffer *ptr);
  void stabilize(int bn);
  char* first_message(int bn);
  char* next_message(int bn);
}; // end of class definition
```

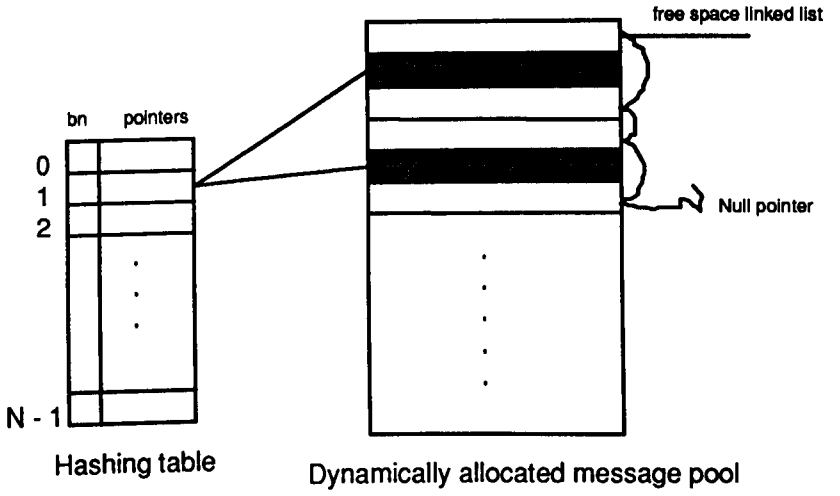


Figure 7.1 - The BM data structure

7.2.3 Time-out Class

The time-out class is the basic class used by the local_time_silence and suspector processes. A time-out object includes a list of Causal Block numbers representing existing Causal Blocks. Once a time-out object is created, one can set a time-out for a Causal Block, switch off a set of time-outs, cancel all time-outs, etc. Below is the simplified definition of the time-out class with its main member functions available.

```
class time-out {
    time-out();
    ~time-out();
    void set(int bn, time long);  // set time to block bn
    switch_off(int bn);           // switch off time-outs up to block bn
    long first_time_out();        // return the "oldest" time_out.
    next_time_out();              // cancel the "oldest" time_out
}
```

7.2.4 Group Naming

Application processes must include the file "group_name.h" and also declare an object of the class multicastList. A multicastList object contains the description of a process group membership. Part of the information to be assigned to the multicastList object is found in the file "group_name" which includes the group_id, unique identifier of members, host addresses of members, etc. The remaining information not present in "group_name.h", is supplied by the application process.

7.2.5 The Configuration File

The configuration file "define.h" must be included by all group members and contains the following information.

Maxmesbuf : the maximum size of message. This is used by the protocol to allocate internal buffers mainly for the BM structure;

Maxnumblocks : this is the maximum number of unstable blocks. This value defines the amount of storage kept for unstable blocks and it is used by the flow control mechanism of Causal Blocks;

Maxmespool : this value defines the maximum number of unstable messages kept in the second level of the BM structure(the dynamic message pool);

MaxGroupSize : this is the maximum group size allowed;

Clockperiod : this is number of milliseconds in between clock ticks for the time-out control[5].

Delta : this is estimated transmission delay time; and

MaxLocalTS : this is the local time silence timeout.

---

[5] see the clock_ticks process description.

7.2.6 The Newtop Class

To use Newtop, the application process simply declares a variable of class Newtop, supplying a multicastList object with the group membership. The two Newtop member functions *send* and *receive* are then used by the application process to send and receive messages that are totally ordered[6]. The constructor of the class Newtop "forks" the parallel processes that will cooperate in order to deliver messages properly. Below we show the simplified C++ code used by an application process to use Newtop.

```
// simplified code of the application process using Newtop.
...
#include <Newtop.h>                  // this is the code of the Newtop class.
#include <groupMembership.h>  // this file contains the details of the group membership
                                     //processes: host addresses, unique identifiers, etc.
...
main() {
  char message[500];          // this is the buffer for the application messages
  multicastList GroupList;   // this object will contains the group membership description
  ...
  GroupList.computation_type = 'c';  // it sends and receives "NewTop" ordered messages
  GroupList.myself = 0;      // 0 is the first process of  groupMembership list
   ...
  NewTop talk(GroupList);   // instanciate a NewTop object
  ...
      // send a message asynchronously
      talk.send(message, messageSize);
      ...
      // receive a message
      messageSize = talk.receive(message);   // it blocks waiting for a message
      ...
  }
```

When the class Newtop is instantiated by an application process, all processes that cooperate to implement the Newtop protocol are created[7], as well as, the corresponding message-queues. A process communicates to another process by putting a message in its queue. For instance, if the receiver process wants to communicate with the deliver process, it sends a message to the deliver process queue, *deliver_Q*. The communication between all processes makes a communication graph where processes are nodes and actual communication through process

[6] messages are delivered respecting causality and in the same global order in all destinations.
[7] This is done by using the "fork" system call.

message-queues, edges (figure 7.2). The system has been carefully designed so as to avoid cycles in the graph which could lead the execution of the protocol to a deadlock. Although each message-queue can keep a number of messages, the inclusion of a new message in a queue will depend on the availability of space for messages in the operating system kernel. To avoid problems of overloading some processes (by having too many messages in its queue) whereas blocking other processes waiting for message space, a flow control mechanism has been used that waits for a number of messages to be consumed before putting new messages in the queue. For instance, the application process can only put one message at once in the *transmitter queue.*

When the *send* operation of a Newtop object is executed, the message to be transmitted is passed to the transmitter process through its queue, the *transmitter_Q*. The *receive* operation when executed will remove a message from the application queue, application_Q. If there is no message available, the application process blocks (without consuming CPU time) waiting for a message to be put in *application_Q*. In the following sub-sections, we will briefly describe all cooperating processes involved in the execution of the Newtop protocol (receiver, deliver, local_time_silence, suspector, transmitter, and the clock_ticks processes).

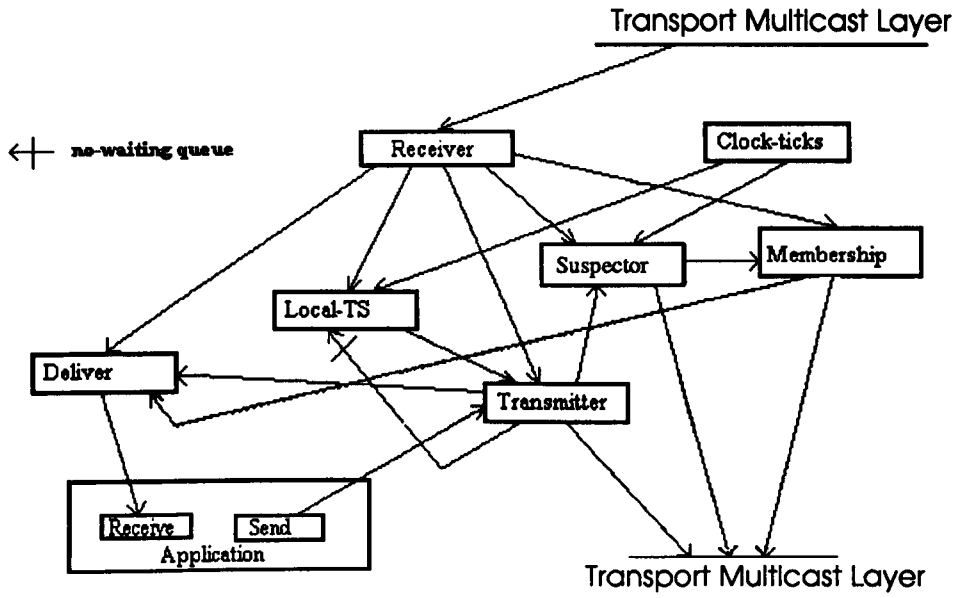


Figure 7.2 - The Newtop Protocol

7.2.7 The Receiver Process

The receiver process receives messages from the transport layer addressed to the a given application process. When a message m is received, the receiver process pass m to the deliver process, and the header of m to the processes local_time_silence, suspector, transmitter, and the membership process.

7.2.8 The Transmitter Process.

The transmitter process can get two types of messages: null and non-null. Non-null messages come from the application process which puts the message it produces in the transmitter queue, transmitter_Q. The local_time_silence process prepares and puts a null message with the largest block-number known at the time the null message is produced. By the time the transmitter process gets this message, some non-null messages may have been produced such that the null message is no longer necessary (i.e, the local BC has increased larger than the null message block-number). To avoid this situation, the transmitter process maintains the block-number of the last transmitted message and discards any null message that has got block-number smaller

than that value. Messages (null and non-null) to be transmitted are then passed to the transport multicast layer, after being duly block-numbered.

The transmitter process also carries out the flow control mechanism of Causal Blocks (see chapter 8). The information necessary for this work, besides messages transmitted, is obtained from the receiver process which puts into the transmitter queue the header part of messages received. Before transmitting a message m with block-number m.b, the transmitter process also prepares the fields m.lcb and m.lsb, the last complete block and the last stable block, respectively. After transmitting a message, the transmitter process puts the message header in the queues of the deliver and suspector processes.

7.2.9 The Deliver Process

The deliver process gets messages from the receiver process and store them in the BM structure (see section 7.3.2). The header part of all messages transmitted (non-null and null) are also put in the deliver process queue by the transmitter process. The deliver process maintains a LRV vector (see section 3.5). By computing the minimum value in LRV, the deliver process figures out complete blocks. Messages from complete blocks are then delivered to the application process (i.e the messages are put into the application process queue).

To compute stable blocks, the deliver process maintains a vector called the Stability Vector (VS) with one entry per process in the group. $SV_i[j]$ represents what process p_i knows about the last complete block number at p_j. When the deliver process detects a block stable (i.e with block number b $\leq$ minimum value in SV), it executes the operation stabilize(b) of the BM structure, freeing the space occupied by block b.

7.2.10 The Clock-Ticks Process

Time-outs in Newtop are implemented based on a clock that ticks each CLOCKPERIOD[8] milliseconds. In order to achieve that, the clock-ticks process is repeatedly put to sleep for CLOCKPERIOD milliseconds using a UNIX system call. Whenever the process wakes up it sends (via message queue IPC) a signal to the local_time_silence and suspector processes that then increment their own local CLOCK. A signal to one of these process is only sent after the previous signal has been consumed, that is, the signaling message has been consumed.

7.2.11 The Local_time_silence Process

This process gets messages from the receiver process, transmitter process, and from the clock_ticks process. When this process is first started by the Newtop class constructor, the C++ long variable CLOCK is initialized with zero; and it is incremented each time an IPC message arrives from the clock_ticks process. At the constructor execution time it is also instantiated an object of the class time-out with the name LT (Local Time_silence). Assume that b' is the block-number of the last transmitted message. Then, whenever a new Causal Block b is created as a consequence of received messages, time-outs are set (with the current value of CLOCK plus the localTS value, a multiple of CLOCKPERIOD, present in the "define.h" file) to all blocks b", $b' < b" \leq b$ using the operation LT.set(b, CLOCK + localTS). When a new message is transmitted, if its block-number, say b, is larger than the largest time-out block-number, say b', then all time-outs are canceled (LT.cancell(b)). Otherwise, all time-outs in the interval (b', b] are canceled using the operation LT.switch_off(b). Time-outs are checked each CLOCKPERIOD milliseconds and when one of them expires, a null-message is sent to the transmitter process, that is, a null-message is put in the transmitter process queue to be transmitted.

---

[8] CLOCKPERIOD is defined in the configuration file :define.h".

7.2.12 The Suspector Process

The suspector process gets messages from the receiver, the transmitter, and the clock_ticks processes. Its function is to suspect processes of having crashed. As in the local_time_silence process, it implements a clock that ticks each CLOCKPERIOD, and, as it was the case for the deliver process, it maintains the LRV vector to detect complete blocks. When the suspector process starts, it is created the RT (Remote Time_silence) object of time-out class. Whenever a new Causal Block b is created due to messages obtained from the suspector queue, a time-out is set to b using the operation RT.set(b, CLOCK + localTS + 2 * Delta). The values of localTS and Delta are the ones present in the configuration file "define.h". When a new block get complete, say LCB, all time-outs up to LCB are canceled with the operation RT.cancel(LCB). Finally, when a time-out expires (the oldest time-out), a *suspicion message* is prepared and sent to the local membership process (via the its message queue) as well as multicast to the remote membership processes (i.e the *suspicion message* is put into the *sending message queue* to be transmitted by the transport layer). The suspector process maintains other two vectors with one entry per group member, the functioning and suspector vectors. When the process group is created, the entries of functioning is set to one and the entries of suspected to zero. When a process P_i is declared crashed, functioning[i] is set to 0, and when a process P_i is suspected, suspected[i] is set to 1. Messages from crashed or suspected processes are ignored by the suspector. Suspicion on a process p_i is removed (i.e suspected[i] is re-set to 0) when a *refute suspicion message* is received from a remote membership process (see next section).

7.2.13 The Membership Process.

The membership process works as an arbiter that based on the *suspicion messages* sent by the suspector processes (one per group member), decides if an over-silent (or suspected) process has crashed or not. In case of a crash indication, the crashed

process is excluded from the group membership. The membership process maintain the following main structures :

```
typedef {
    int lbn; // last block number from the over-silent (suspected) process.
    int agreed[MAXGROUPSIZE]; // indicates the processes that have agreed on lbn.
} suspicion;
suspicion suspicions[MAXGROUPSIZE]; // keeps all crash suspicions
suspected[MAXGROUPSIZE];
functioning[MAXGROUPSIZE];
LRV[MAXGROUPSIZE];
```

The header of messages sent or received by the application process are passed to the membership process through its message queue. These messages are represented in the structure LRV for working out complete blocks. When a *suspicion message* is received from a *suspector process*, either a refute message will be generated and multicast to the group or it will be represented in the suspicions set. When all unsuspected (not suspected) processes have suspicion messages about a given (pid, lbn) pair represented in the suspicions set, the process identified by pid is removed from the group membership by setting functioning[pid] to 0.

7.3 Experimental Results

In this section we will comment on the performances results obtained from a series of experiments where various parameters have been monitored during the execution of the Newtop. In section 7.4.1, we explain some terminology used and describe the different experiments we have carried out. In section 7.4.2, we will comment on the experimental data collected.

7.3.1 Performance Measures

Due to the unpredictability of transmission delays in asynchronous systems, message delivery delay of a total order multicast protocol for such systems is measured in terms of the number of extra messages transmitted so as to deliver a multicast. In the case of Newtop, these messages are null messages generated by the time-silence mechanism in order to complete "old" incomplete blocks. Another

relevant performance measure is the message space overhead, and in the case of our protocol, this overhead is constant and very small (basically, the message block-number).

In Newtop, messages are delivered on the basis of block completion. This means that messages to complete a given block have to be sent by all group members before that block can be delivered. So, the activity (rate of message transmission) of the group members will determine how fast blocks get complete, and consequently, messages delivered. The Local Time-Silence mechanism guarantees that messages are always delivered despite the inactivity of some group members but on the expense of null messages transmitted.

To analyze the behavior of our protocol we have carried out experiments for the two extreme case scenarios in terms of processes activity. That is, the worst case scenario, when only one process sends messages to the group whereas the others remain silent, and the best case scenario, when all processes are active (sending messages to the group). By analyzing the behavior of the protocol on these conditions, one can get a good insight of its performance in the presence of distinct scenarios.

Experiments have been run for different group configurations, varying parameters such as the inter-message transmission time period and the local time silence timeout. Before presenting the figures, let us explain briefly some important concepts and notations.

- CLOCKPERIOD -

In the current implementation of Newtop, timeouts of the local time silence and suspector processes are verified in intervals of CLOCKPERIOD milliseconds. In fact, a CLOCK have been implemented that progresses (ticks) each CLOCKPERIOD milliseconds. The timeout associated with an event will be computed taking the current value of this CLOCK plus an integer number (representing a number of ticks) and it will expire when the current value of CLOCK reaches that value.

- Inter-message transmission time period -

This is the amount of time (in milliseconds) an active process waits before sending a message to the group. In other words, messages are sent by active processes in intervals of "Inter-message transmission time period" milliseconds.

- Local Time Silence (Local TS) -

This is the timeout for sending null messages. The value of Local TS will be a multiple of CLOCKPERIOD. For instance, if CLOCKPERIOD is equal to 50 and Local TS is equal to 4, then, the local time silence process will wait approximately 200 milliseconds until timeouts. Notice that the actual time for the example cited above will vary between 150 and 200 milliseconds, depending on when the event occurred, just before incrementing CLOCK or just after incrementing it, respectively.

- Maximum number of unstable blocks (Max# unstable) -

This value represents the maximum number of blocks found unstable during a given experiment.

- Average delay overhead (Avr. delay ov.) -

The delivery delay overhead for our protocol, is the time elapsed between the receipt of a message m by the receiver process and the delivery of that message by the deliver process. When m is received from the multicast layer by the receiver process, it is timestamped with the current wall clock value and placed in BM. Then, when it is delivered (put the in application message queue), the current wall clock value is taken again and its value minus m's timestamp is taken as the delivery delay overhead of m. For a given experiment, the average delay overhead of messages delivered by a group member is the sum of all message delay overheads divided by the number of messages delivered. Finally, the average delay overhead showed in the figures is the sum of the average delay overheads of all group members divided by the number of members.

- THROUGHPUT -

Say that in a given experiment N messages are sent to the group. Assume that all group members after delivering those messages send a reply to the group confirming the delivery of the N messages. Also, assume that USER0 is the group member that

sends the very first message in the group. Say that T is the wall clock time elapsed between just before sending the very first message and the delivery of the N messages plus all replies at USER0. Thus, the throughput for that experiment is N divided by T.

7.3.1.1 The 1-active Experiment

In this experiment, the application process USER0 sends 1000 messages of 32 bytes to the group. The other processes deliver 1000 messages and after that send a reply to the group confirming the delivery. There are two computations related to the application process USER0 (the sender computation and the receiver computation). The other processes only have one computation that sends and receives messages. We present below the simplified code for the application processes.

USER0 - The sender computation

```
timer t;
groupList GL;          // groupList is the class that defines the initial group configuration.
GL.num_participants = N;
GL.myId = "myId";
// other initializations
NewTop talk(GL);    // NewTop is the C++ class that implements  the NewTop protocol.
long t0 = t.current_time();
for (int i = 0; i < 1000; i++) talk.send(t0);
...
```

USER0 - The receiver computation

```
timer t;
groupList GL;
GL.num_paticipants = N;
NewTop talk(GL);
// other initializations
for (int i = 0; i < 1000; i++) talk.receive(m);
// get the replies
for (int i = 0; i < GL.num_participants - 1; i++) talk.receive(m);
// work out the elapsed time
long ftime = t.current_time();
long delay = ftime - m;  // notice that m is the initial time transmitted by USER0 in the sender
                         // computation.
...
```

Other processes - sender/receiver computation

```
groupList GL;
GL.num_paticipants = N;
NewTop talk(GL);
. . .
for (int i=0; i<1000; i++) talk.receive(m);
// send reply
talk.send(m);
// deliver all replies
for (int i=0; i < GL.num_participants - 1; i++) talk.receive(m);
. . .
```

7.3.1.2 The all-active Experiment

In this experiment, each application process sends 1000 messages of 32 bytes to the group. The application process identified as USER0 sends a signal (using sockets) to the other members begin to transmit the messages. All processes except USER0 send a reply after delivering all transmitted messages (1000 * number of members). We show below the simplified code for the application processes. There are two computations for each application process: the sender and the receiver computations.

USER0 - The sender computation

```
timer t;
groupList GL;          // groupList is the class that defines the
                       // initial group configuration.
GL.num_participants = N;
GL.myId = "myId";
. . .
NewTop talk(GL);              // NewTop is the C++ class that implements  the NewTop protocol.
long t0 = t.current_time();
// sends a sign to all group members to begin the transmission (using
// TCP/IP sockets.
for (int i = 0; i < 1000; i++) talk.send(t0);
 . . .
```

USER0 - The receiver computation

```
timer t;
groupList GL;
GL.num_paticipants = N;
NewTop talk(GL);
. . .
long t0;
talk.receive(t0);  // Notice that the sender computation of USER0 transmits the initial time.
for (int i = 1; i < 1000 * N; i++) talk.receive(m);
// get the replies
for (int i = 0; i < N - 1; i++) talk.receive(m);
// work out the elapsed time
long ftime = t.current_time();
long delay = ftime - t0;
 . . .
```

Other processes - sender computation

```
timer t;
groupList GL;          // groupList is the class that defines the  initial group configuration.
GL.num_participants = N;
GL.myId = "myId";
. . .
NewTop talk(GL);       // NewTop is the C++ class that implements the NewTop protocol.

// waits the sign to begin to transmit ...

// multicast 1000 messages
for (int i = 0; i < 1000; i++) talk.send(t0);
. . .
```

The receiver computation

```
timer t;
groupList GL;
GL.num_participants = N;
NewTop talk(GL);
...
for (int i = 0; i < 1000 * N; i++) talk.receive(m);
// get the replies
for (int i = 0; i < N - 1; i++) talk.receive(m);
 ...
```

7.3.2 Commenting on Performance Results

Figures 7.3, 7.4, and 7.5 show the data collected for the 1-active 3-process group experiment, when the inter-message transmission time period was set to 100 msecs. We have measured three different performance parameters against different time-silence timeouts values. Figure 7.3 shows the maximum number of unstable blocks present at all members during the experiment. Notice that the number of unstable blocks tends to grow for increasing time-silence timeouts. Figure 7.4 shows the total number of null messages transmitted by all 3 group members during the experiment. This number tends to decrease for increasing time-silence timeouts. Figure 7.5 shows the average delay overhead observed during the experiment. The average delay overhead increased for increasing time-silence timeouts. The behaviour observed in the experiments of figures 7.3, 7.4, and 7.5 is present in all experiments we have carried out. Therefore, choosing the appropriate value for the time-silence timeout can give to a system designer the desired performance for specific application/system requirements. Shorter time-silences, gives better delay overhead and keeps the number of unstable blocks smaller. On the other hand, the number of null messages transmitted increases.

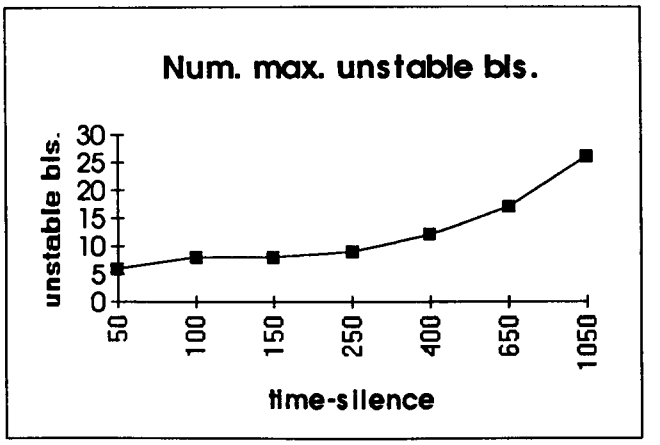


Figure 7.3 - The 1-active 3-member group experiment. Maximum number of unstable blocks.

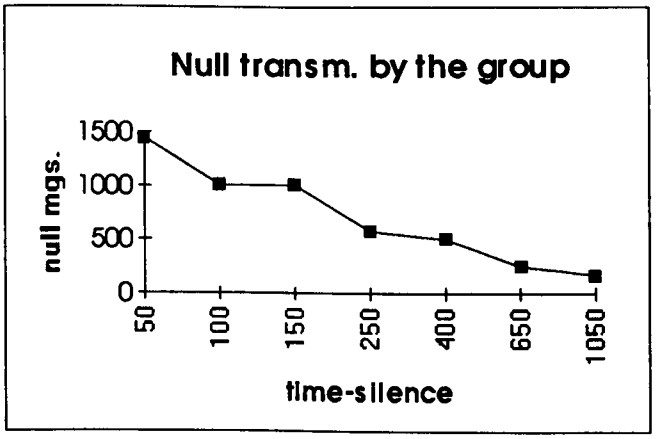


Figure 7.4 - The 1-active 3-member group experiment. Null messages transmitted by all group members.

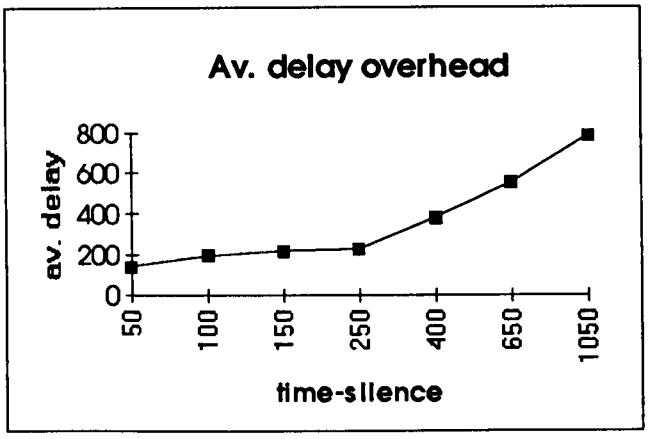


Figure 7.5 - The 1-active 3-member group experiment. The average delay overhead.

Figures 7,6, 7.7, and 7.8 show data collected for the 1-active experiment when inter-message transmission time period was set to 200 msecs. The graphs in figures 7,6, 7.7, and 7.8 show maximum unstable blocks, average number of null messages transmitted per inactive process (the receivers), and average delay overhead, respectively, for time-silence timeouts from 50 to 1050 msecs. We have run the experiments for different group configurations, from 2 to 6 group members. In each of the graphs there is a curve representing the data for a given group configuration (the group size appears on the legend). Group processes were spread over three workstations. Notice the same pattern present in the previous experiment is also present here, despite the number of group members. That is, increasing the time-silence timeout makes the number of unstable blocks to grow, increases the delivery delay, and decreases the number of null messages transmitted.

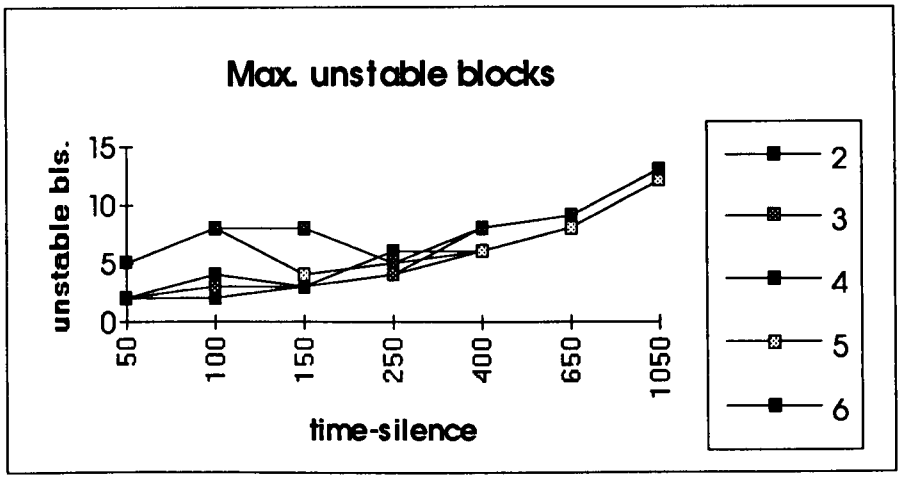


Figure 7.6. The 1-active experiments for different group configurations. Maximum number of unstable blocks.

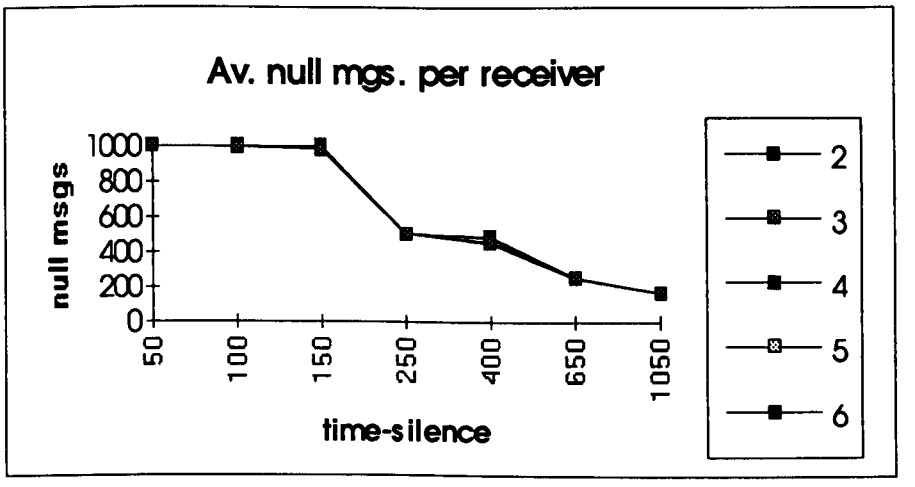


Figure 7.7 - The1-active experiments for different group configurations. Average number of messages transmitted per inactive process.

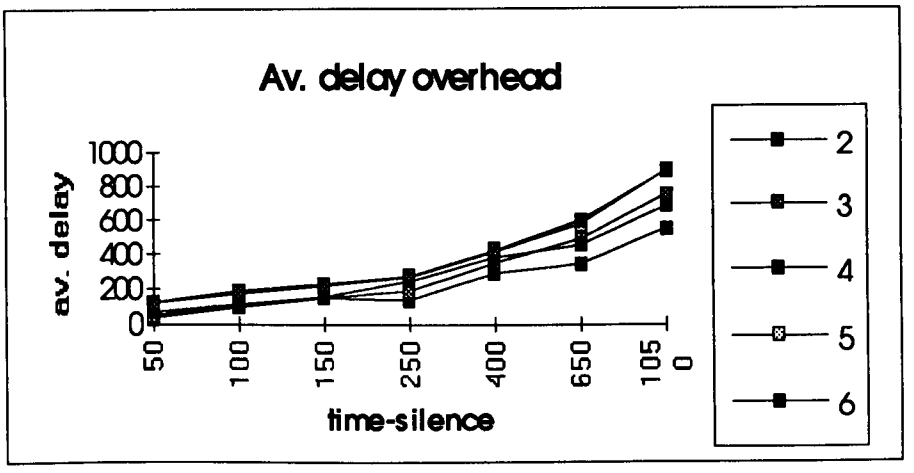


Figure 7.8 - The1-active experiments for different group configurations. The average delay overhead.

Figure 7.9, 7.10, 7.11, and 7.12 show data collected from the 1-active experiment when time-silence was fixed to 100 msecs. For this experiment, we have varied the inter-message transmission time period from 400 msecs down to 6 msecs. Observe that for smaller inter-message transmission periods, the number of unstable blocks increases and the number of null messages decreases. For the same variation, the average delay overhead increases. This increase of the average delay overhead is due to the fact that a larger number of unstable blocks imposes extra delay overhead on messages waiting for block completion. Thus, adding to the overall average message delivery delay overhead. Finally, notice that despite the increase on average delay, the numbers of messages delivered per second (throughput) increased for smaller inter-message delivery time. The explanation for this is that when the inter-message message transmission period in small (so, transmission rate is high), several blocks are

delivered at once. Thus, as long as an application can afford buffers, the existence of incomplete blocks will not affect the overall system performance.

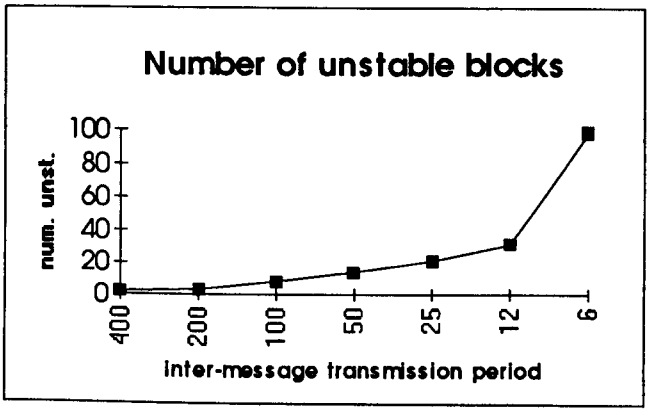


Figure 7.9. The 1-active 3-member group experiment. Maximum number of unstable blocks.

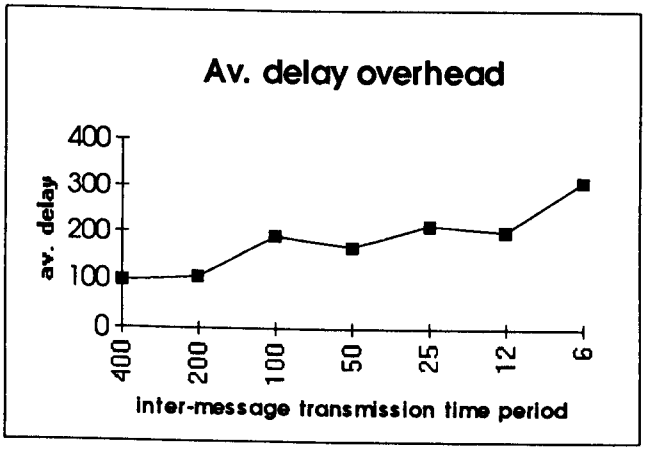


Figure 7.10. The 1-active 3-member group experiment. Average delay overhead.

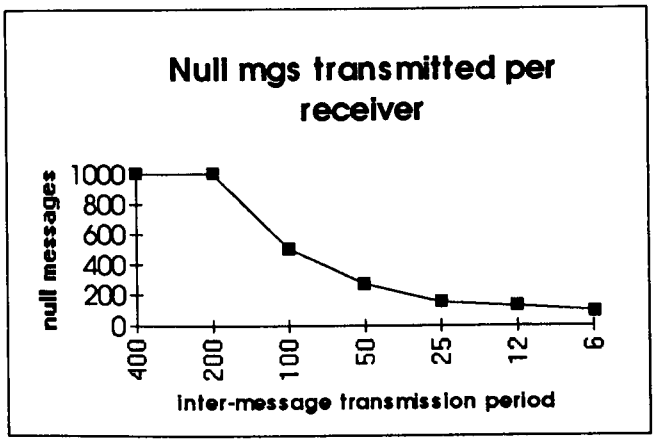


Figure 7.11. The 1-active 3-member group experiment. Average number of null messages transmitted per receiver.

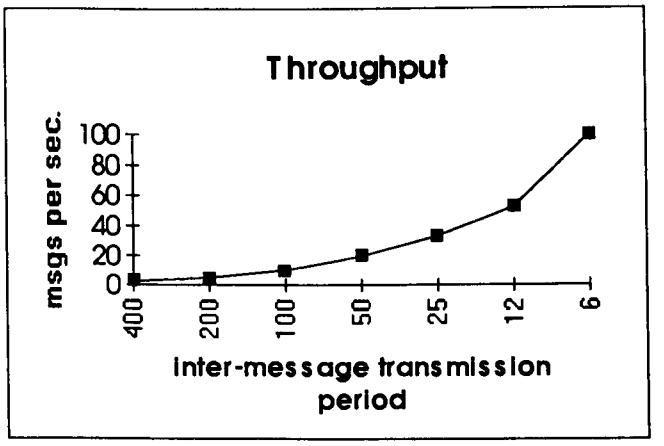


Figure 7.12. The 1-active 3-member group experiment. Throughput: number of messages delivered per second.

We now comment on the data for the all-active experiment. Figures 7.13, 7.14, and 7.15 show the data collected for the all-active 3-member group experiment when

the inter-message transmission time period was set to 400 msecs. These figures show the maximum number of unstable blocks, total number of null messages transmitted, and average delay overhead obtained during the experiment, respectively. Notice that since all processes are active, if we had a time-silence timeout larger then the inter-message transmission time period, no null messages would be transmitted. So, for this experiment we have used time-silences shorter than 400 msecs. We varied them from 350 to 100 msecs. Notice that, different from the 1-active experiment, the use of varied time-silences did not cause much impact on the number of unstable blocks and average delay overhead. The only noticeable impact was on the number of null messages transmitted.

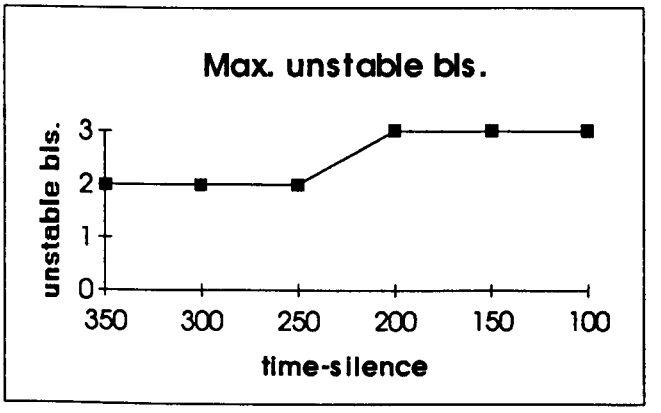


Figure 7.13 - The all-active 3-member group experiment. Maximum number of unstable blocks.

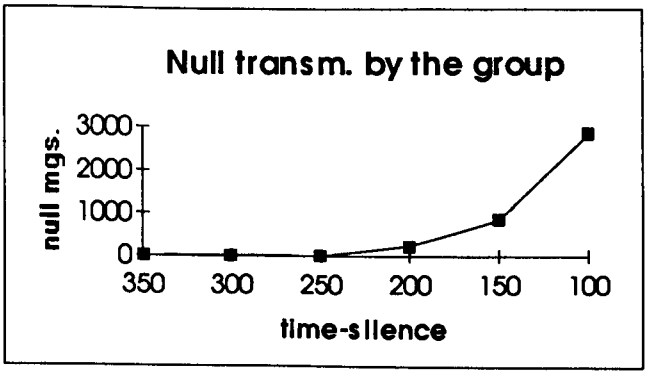


Figure 7.14 - The all-active 3-member group experiment. The total number of null messages transmitted.

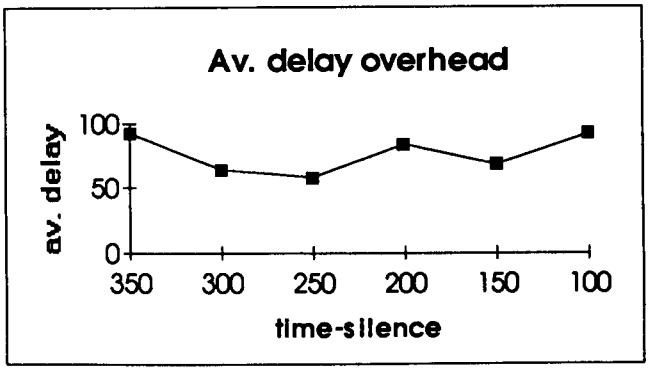


Figure 7.15 - The all-active 3-member group experiment. The average delay overhead.

The next three figures (7.16, 7.17, and 7.18) show data collected from the all-active experiment when the inter-message transmission time period was set to 500 msecs for time-silences values from 450 down to 200 msecs. We have run the experiment for different group sizes, from 2 to 6 group members. Figures 7.16, 7.17, and 7.18 show the maximum number of unstable blocks, the average delay overhead, and the number of null messages transmitted by the group, respectively, during the experiment. Notice that, as it occurred for the all-active 3-member group experiment, the variation on time-silence timeouts did not cause a great impact on the number of unstable blocks and average delay overhead (although the average delay overhead slightly decreased for smaller time-silence timeouts). Similarly, the number of null messages transmitted increased for shorter time-silence timeouts. Observe that although all application processes for this experiment transmit messages in the same rate, they work asynchronously (all processes run in a multi-task and mult-user environment). Therefore, messages to complete a given block will not arrive (or be transmitted) at the same physical time, causing null messages be transmitted by the time-silence mechanism.

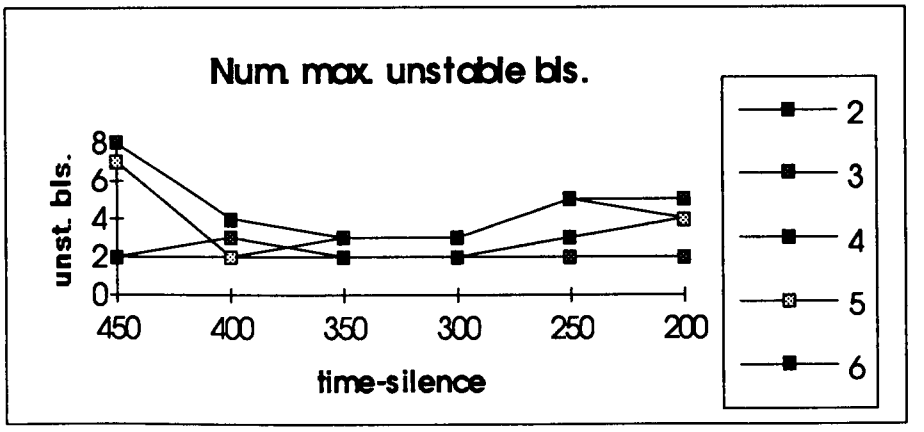


Figure 7.16 - The all-active experiment for varied group sizes. Maximum number of unstable blocks.

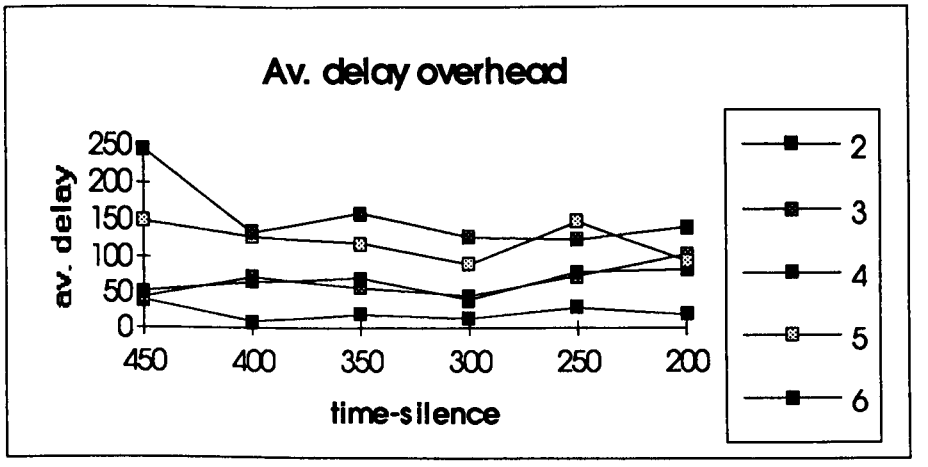


Figure 7.17 - The all-active experiment for varied group sizes. The average delay overhead.

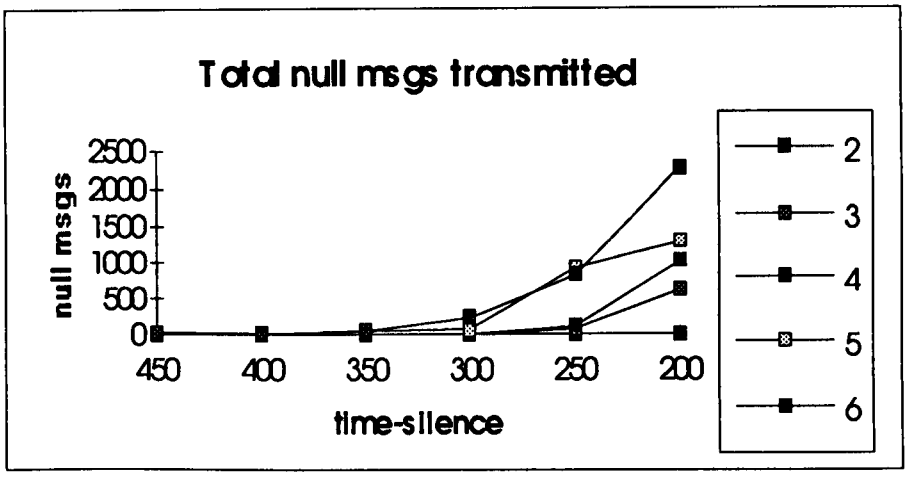


Figure 7.18 - The all-active experiment for varied group sizes. Total null messages transmitted by the group.

7.4 Conclusions

We have described the implementation of the Newtop protocol. The implementation described has been tested over a set of networked UNIX sparc stations. We have run experiments to evaluate the performance of Newtop under varied group configurations, transmission rates, and time-silence timeouts, when processes crashes were not considered. The value set for the time-silence timeout is determinant on the number of null messages transmitted. When only one group member is transmitting messages, choosing shorter time-silence timeouts will on one hand cause a larger number of null messages be transmitted, but on the other hand, it will reduce the message delay overhead and the number of unstable blocks produced. When all group members are actively transmitting messages at the same rate, varying the time-silence timeout does not cause a great impact on the number of unstable

blocks and the average delay overhead improved only slightly for shorter time-silence timeouts. So, choosing the appropriate time-silence value is a key point for tuning the behaviour of Newtop to specific application and system requirements such as local buffers (maximum number of unstable blocks), delay overhead, and consumption of transmission bandwith (extra null messages transmitted).

During the first implementation tests, the execution of the protocol was sometimes interrupted due to the running out of local buffers for unstable messages. Typically, this happened when we combined small inter-message transmission times with large time-silence timeouts (localTS). That means, receiver processes were not sending stability information quick enough so that blocks could be stabilised at the sender's side. This has motivated us to develop the flow control mechanism that we will present in the following chapter.

Chapter 8 - A Flow Control Scheme for Fault-Tolerant Multicast Protocols

Fault-tolerant group communication protocols require that a transmitted/received message be kept locally for possible retransmission until certain ordering and reliability conditions have been satisfied at all the members of the group. If we assume variable (potentially unbounded) transmission and message processing delays - in what would be an asynchronous distributed system - then the number of such 'unstable' messages at processes may grow indefinitely, leading to the possibility of buffer overflows; hence the need for a flow control mechanism. This chapter addresses the problem of flow control in an asynchronous, total order group communication protocol. We have developed a mechanism that guarantees that the number of unstable messages does not exceed the stated bound, thus preventing buffer overflows. Our mechanism is *safe* (no buffer overflows) as well as *lively* (a sender will be eventually permitted to send).

The flow control mechanism to be described here has been designed and implemented for Newtop (see chapters 4 and 7). In a symmetric total order protocol, such as Newtop, when a message is received at a node, its delivery to the local process(es) may be delayed, as there may be concurrent or causal related messages from other members of the group in transit. Once it is certain that all such messages have been received, then these messages can be delivered. That is, once a Causal Block is detected to be complete (there are no more distinct messages to be received with the same block-number), the received messages can be delivered to the process in some fixed order. In order to ensure that Causal Blocks eventually complete, as has been described in section 4.3, we employ a *time-silence* mechanism that ensures that a process transmits a message now and then (only a null message, if a given process has no other message to transmit).

Consider now that a process multicasting a message crashes (or gets disconnected) in the middle of the multicast, such that only some of the members receive the message. In this situation, the time-silence mechanism alone will not be able to ensure block completion. For this purpose, Newtop (in common with other protocols, e.g., [Amir92b, Mishra93]) provides another liveness mechanism: each process is associated with a local *group-view* process that can execute a *membership protocol* (see chapter 6) with its counterparts to reach agreement on the membership of the group. Thus, if the group-view process of p_i *suspects* a failure of some remote process (p_j) that does not seem to be responding, then the group-view process can initiate a membership agreement on p_j, the outcome of which is that either processes agree to eliminate p_j from the group, with an agreement on the last message sent by p_j, or p_j continues to be a member of the group and p_i is able to retrieve missing messages of p_j. Thus, even if a Causal Block is complete at a process, it is necessary to keep the messages with that block-number for a while (as it may be asked to supply some of those messages to others). Once it is known that a given block is complete at all the members of the group, only then the messages with that block-number can be discarded after delivery (such messages are said to be stable). Our flow control mechanism is therefore based on piggybacking block completion and message stability information on normal messages.

Fortunately, in order to explain the basic principles behind our flow control mechanism, we do not need to know how the membership service (or even the time-silence mechanism) actually works: it is sufficient to assume the existence of liveness mechanisms that ensure that a given Causal Block will eventually complete. Indeed, we will ignore process failures altogether, and simply assume that there is a group $g = \{p_1, p_2, \ldots, p_n\}$ and that every member process p_i, $1 \leq i \leq n$, knows the membership, and member processes communicate with other members only by multicasting to the membership of the group. We assume the existence of a message transport layer permitting best effort uncorrupted and sequenced message transmission between a

sender and destination processes, if the processes are alive and the destination processes are not partitioned (either due to physical network failure or message congestion) from the sender; no assumption about message transmission time is made. We assume that all processes have identical, fixed amount of local buffer spaces available for storing received messages and housekeeping information. A sender blocks if the sending of a message can cause the buffer space to overflow at any of the member processes. For the sake of simplicity in exposition, we will assume that all messages have the same fixed size. If a process belongs to multiple groups (Newtop has been designed to deal with this case), then our flow control mechanism can be individually operated for each of the groups the process belongs to. To achieve that, a separate Block Counter can be created for each group a multi-group process p_i belongs to (transmitted messages would carry two block-numbers: one intended for message delivery and another for flow control). Causal Blocks related with the Block Counter used for flow control would however not be created and no message space alocated to them. Their existence can be represented by a single vector called the Last Received Vector (LRV) that provides an efficient way of block completion detection (see section 3.2). Since the flow control operates individually for each of the groups, we need only consider the case of a single group. So, in what follows, we will assume Newtop in a uni-group environment such that the same Block Counter BC will be used for both message delivery and the flow control mechanism, and therefore, transmitted messages will be timestamped with only one block-number.

8.1 Flow Control in Newtop

Let us assume that the maximum size of the Block Matrix has been fixed to N. As stated before, it is necessary to ensure that a process can always retrieve a missing message from another functioning member process. This in turn means that we require a mechanism that enables a process to safely discard a received message. To develop such a mechanism, we use the concept of *message stability* (first defined in section 4.4.1 and repeated here).

Message Stability: A Causal Block $BM_i[\beta]$ becomes stable if it is known to be complete in all the processes in the group. The messages represented in $BM_i[\beta]$ will also be termed stable.

The flow control mechanism for Newtop operates at the level of Causal Blocks and ensures that multicasting of a message will not cause the limit of no more than N unstable blocks at any process to be exceeded. Blocks become stable in increasing order of block-numbers: if block $BM_i[\beta]$ is stable, then all blocks $BM_i[\beta_0]$, $\beta_0 < \beta$, will also be stable. Once a Causal Block becomes stable at p_i, p_i "knows" that the corresponding messages have been received by all the members. Stability information is passed together with transmitted messages. That is, when a message *m* is transmitted by p_i, a field *m.lcb* is used to represent the number of the *largest complete block* in BM_i. To compute stable blocks, each process p_i, maintains a vector called SV_i (Stability Vector). At process p_i, $SV_i[j]$ represents the largest complete block at p_j. If $\min(SV_i)$ represents the minimum value in SV_i, then all blocks $BM_i[\beta]$, $\beta \leq \min(SV_i)$ will be stable. For the sake of fault tolerance, a block that is not stable, and the messages represented in it are not discarded from the local storage of processes. Once a stable block has been discarded, the freed space becomes available for reuse. The flow control mechanism developed here is based on a stronger stability condition, called *n^2- stability*.

Message n^2-stability: Causal Block with number β is n^2-stable if $BM_i[\beta]$ is stable at all $p_i, 1 \leq i \leq n$.

An n^2-stable block by definition does not exist in the BM of any group member. That is, if β is an n^2-stable block, then, all group members would have discarded it from their respective BMs. As was the case for stability, if a block β' is n^2-stable then all $\beta \leq \beta'$ will also be n^2-stable. Whenever a new Causal Block, say with number β, is to be created by the sender of a message *m* ($m.b = \beta$), the sender first verifies the condition *fl1*:

fl1: Causal Block with number β - N is n^2-stable.

If *fl1* is true, *m* can be sent with the assurance that the slot addressed by *m.b* - N will be free in the BM of every recipient; otherwise, the sender will wait for the condition *fl1* to become true. In order to compute n^2-stability, each process maintains a vector of size n called the *n^2-stability vector* (NNS). At process p_i, $NNS_i[j]$ records the number of the largest stable block at process p_j. If $\min(NNS_i)$ is the minimum value represented in NNS_i, then all blocks $\beta \leq \min(NNS_i)$, are n^2-stable. A new field, *lsb* (for largest stable block), is added to a message to disseminate n^2-stability information: whenever *m* is transmitted by p_i, the field *m.lsb* will represent the number of the largest stable block at p_i at the time of transmission (it will be helpful to recall here that in the field *m.lcb,* the message also carries the stability information, the number of the largest complete block at the sender).

Although applying condition *fl1* is safe in avoiding overflows, if enforced in isolation, it may lead the sender process to starvation and the whole system to a deadlock. For instance, consider the scenario where a process p_i is about to transmit *m* with block-number β and blocks awaiting condition *fl1* to become true. Also, consider that after a given period of time, all blocks in BM of p_i become complete but no one is stable. Because stability information is transmitted together with messages (null or non-null), p_i will block forever, and soon so will other processes as they must receive information from p_i to stabilise blocks. To avoid this situation we need conditions that guarantee that blocks eventually become n^2-stable. This can be realised if, in addition to *fl1*, the sender also verifies conditions *fl2* and *fl3*:

fl2: Causal Block with number $\beta + 1$ - N is stable; and,

fl3: Causal Block with number $\beta + 2$ - N is complete.

So, whenever a new Causal Block, say with number β, is to be created, the sender blocks if conditions *fl1, fl2 and fl3* are not true. The rationale behind *fl2* and *fl3* is as

follows. By applying condition *fl2*, it is guaranteed that a received message *m* will bring the information that at least block $\beta - N + 1$ is stable at the sender. Since this condition will be followed by all processes, it is guaranteed that when block β completes at p_i, block $\beta - N + 1$ will be n^2-stable, and block $\beta + 2 - N$ is stable (thanks to *fl3*). Thus completion of a block, β, ensures that the first two conditions are true for the next block ($\beta + 1$). Fortunately, the liveness mechanisms of Newtop ensure that a Causal Block eventually completes, so condition *fl3* can always be relied upon to become true. The minimum size of a Block Matrix is therefore three. The proof of correctness (that the three conditions together implement flow control properly for $N \leq 3$, i.e., a BM never overflows and a sender never blocks indefinitely) is given below after an illustrative example. Consider that Block-Counters are initialized by -1 so that the first Causal Block created will have block-number equal to zero.

Figure 8.1 shows the BM of a process, say p_i, where the maximum size of the BM of processes has been set to 8. In figure 8.1(a), blocks 0, 1, and 2 are n^2-stable, block 3 is stable and block 4 is complete. The other blocks (from 5 to 10) have not yet been created (thus, no message space allocated). Given this situation, p_i will be able to send messages with block-numbers 5 to 10 (figure 8.1(b)), but will be blocked for the sending of a message with block-number 11. For this case, the message can only be sent once block 3 become n^2-stable, block 4 becomes stable, and block 5 becomes complete.

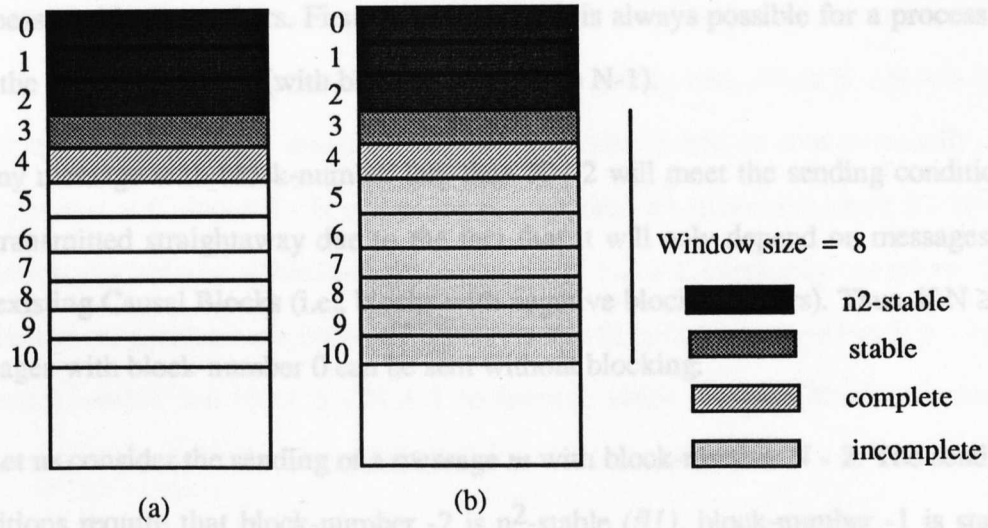


Figure 8.1 - n^2-stable, stable, complete and incomplete Causal Blocks

Correctness proof.

We now prove the correctness of the flow control mechanism, for any $N \geq 3$.

Lemma 1: BM does not overflow (*safety*).

Proof: The proof is by contradiction: suppose that BM_i of a process p_i overflows while trying to represent a received message *m*. This means that a new Causal Block needs to be created but BM_i of p_i has already N unstable blocks. In particular, this means that the slot (*m.b mod* N) in BM_i points to an unstable block. By definition, just before *m* was sent, condition *fl1* had to be true. That is, block-number *m.b* - N was n^2-stable at *m*´s sender; this implies that block-number *m.b* - N was stable at p_i and, consequently, the space reserved for messages with block-number *m.b* - N freed. Therefore, BM_i of p_i could not overflow while trying to represent *m*.

Lemma 2: A sender never blocks indefinitely (*liveness*).

Proof: Assuming that once a Causal Block is created, block completion liveness mechanisms of the protocol will ensure that the block will eventually complete, we will prove the liveness of the sending conditions *fl1*, *fl2*, and *fl3* using induction on

the message block-numbers. First we show that it is always possible for a process to send the first N messages (with block-numbers 0 to N-1).

(i) Any message with block-number less than N - 2 will meet the sending conditions and transmitted straightaway due to the fact that it will only depend on messages of non-existing Causal Blocks (i.e., blocks with negative block numbers). Thus, if $N \geq 3$, messages with block-number 0 can be sent without blocking.

(ii) Let us consider the sending of a message *m* with block-number N - 2. The sending conditions require that block-number -2 is n^2-stable *(fl1)*, block-number -1 is stable *(fl2)*, and block-number 0 is complete *(fl3)*. Since blocks -1 and -2 do not exist, we need only prove that *fl3* will eventually be true. Provided $N \leq 3$, this will be the case since messages with block-number 0 meet the sending conditions, and once created, block 0 will eventually complete.

(iii) Now consider the sending of a message *m* with block number N - 1. The sending conditions require that block number -1 is n^2-stable, block number 0 is stable, and block number 1 is complete. From (ii) we can see that once block number N - 2 completes at the sender, block 0 is bound to become stable (because a message with block number N - 2 will carry the information that block-number $i \leq 0$ is complete at the sender); further, like block 0, block 1 will also eventually complete.

Consider the case when the message to be sent has block number equal to N. We must show that the following will eventually be true: block 0 is n^2-stable, block 1 is stable, and block 2 is complete. From (iii) we can state that once block N - 1 completes at the sender, blocks 0 and 1 are bound to become n^2-stable and stable respectively; further, like block 1, block 2 will also eventually complete.

Finally, consider the sending of a message with block-number $\beta > N$. Since block counter is incremented by one before transmission, a (null or non-null) message with block-number β-1 must have been sent before β. So, by induction hypothesis, the

following was true before the message with block number β - 1 was transmitted: block β - N - 1 was n^2-stable; block β - N was stable; and, block β - N + 1 was complete. To prove that a message with block number β will be sent eventually, we must prove that (i) block β - N will eventually become n^2-stable; (ii) block β - N + 1 will eventually become stable; and (iii) block β - N + 2 will eventually complete. This will be the case, since completion of block β - 1 will bound to cause block β - N to become n^2-stable and block β - N + 1 to become stable. Finally, block completion liveness mechanisms will ensure that block β - N + 2 completes.

Observation:

The application of the sending conditions *fl1*, *fl2*, and *fl3* will not prevent null messages from being eventually transmitted by a process p.

Suppose there is a null message u with block number u.b to be transmitted by a process p. Because null messages are only transmitted to complete existing blocks (see section 4.3), there is at least one non-null message m that was transmitted by a process q with block number m.b equal to u.b. This means that by the time m was sent, all group members had produced enough information together with transmitted messages so as the sending conditions for m could be satisfied. By our transport assumption, we realise that those messages will eventually reach all destinations, including process p, when then, the sending conditions for u will be met.

8.2 Experimental Results

The flow control mechanism presented in this chapter has been implemented and tested on the Newtop protocol (see chapter 7). In the implementation of Newtop, the BM matrix is regarded just as a conceptual entity, with the necessary information therein being encoded in a vector of size n, called the Last Received Vector (see section 3.5). Received messages with the same block-number are kept in a linked list of buffers, with an entry in a hash table of size N containing the pointer to the head of

the list. Recall that N is the size of the Block Matrix, so N is also the maximum permissible number of unstable blocks; therefore the maximum number of buffers required is nN. The addressing function of the hash table takes a message block-number and generates an address in the interval [0, N-1]. Messages with block-number β are discarded only after Causal Block with block-number β becomes stable; after this the hashed entry in the table becomes available for pointing to messages of a new Causal Block.

We have performed experiments to evaluate the effect of the flow control mechanism on the performance of Newtop. In the experiments we have a process group with six members distributed over three workstations. We consider the case when only one process is sending messages. In this circumstance, the time-silence mechanism of the inactive processes is responsible for the completion of blocks. The time-silence timeout for the experiments was fixed to 50 msecs. The graphs depicted in figures 8.2 and 8.3 are for the sender process. For each run, 1000 messages of 32 bytes each were transmitted. We varied the inter-message transmission time from 400 msecs to 6 msecs and calculated the maximum number of unstable blocks as well as the average delivery delays.

Figures 8.2 and 8.3 graphically show the data collected during the experiments when the flow control mechanism was switched off and on (with N fixed to 50) respectively. It is interesting to note that when the flow control was switched on, besides limiting the number of unstable blocks to 50, the average delivery delay was also reduced for inter-message transmission times of less than and equal to 50 msecs. The explanation for this is that when the transmission rate is higher, a larger number of messages are transmitted without stability information (i.e. the last complete block number) being updated at the sender. This causes the number of incomplete blocks to grow quickly and thus increases the average delay overhead for message delivery.

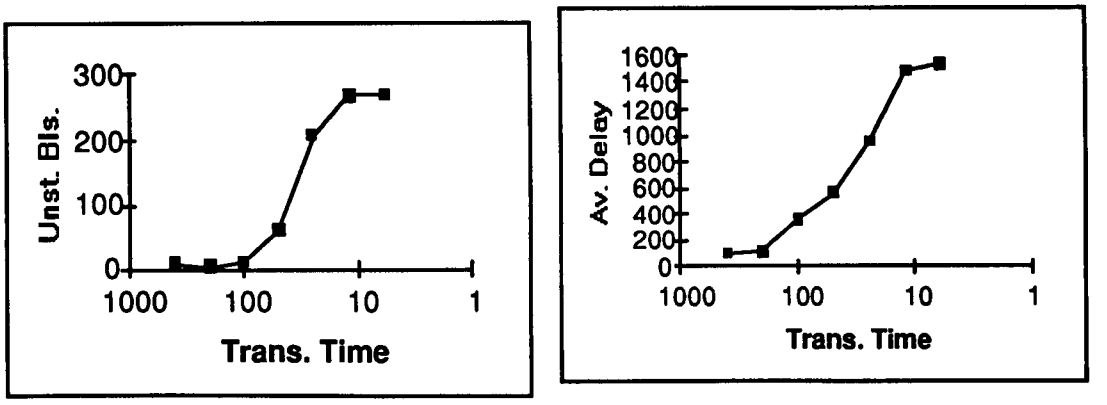


Figure 8.2 - 1-sender and 5-receivers with flow control switched off.

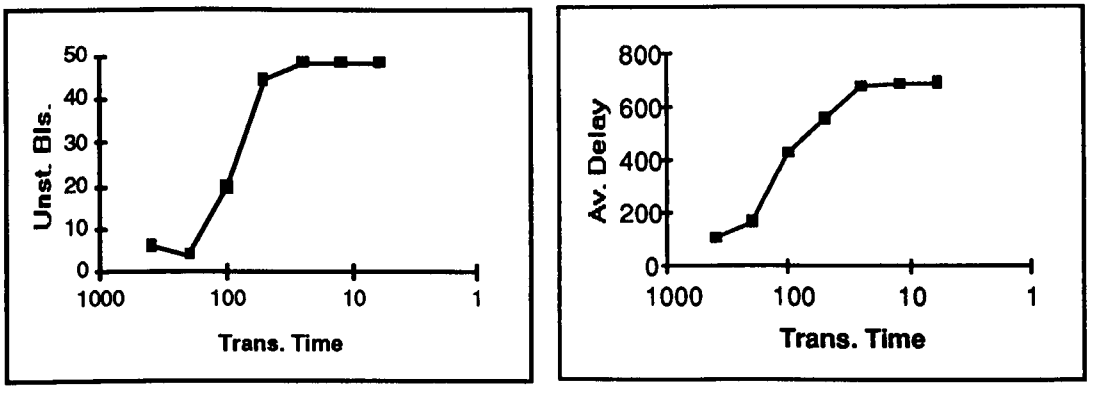


Figure 8.3 - 1-sender and 5-receivers with flow control switched on.

8.3 Related Work

Flow control schemes for point-to-point communication protocols (such as TCP/IP) have been extensively explored in the past [Tanenbaum81]. Usually, these protocols utilise the concept of sliding windows. In a sliding window scheme, transmitted messages are numbered sequentially before being sent to the destination. At any instant, the sender has a "window" on this sequence of messages and have permission from the receiver to send any message within this window. So, if the window extends from n to n + k, the sender can send any message numbered from n to n + k. As the receiver acknowledges the receipt of messages, say w messages, the window slides (moves) forwards in the ordered sequence. Thus, on receiving the

acknowledgments for the w messages, the sender can now send messages numbered from n + w to n + k + w. The sequence numbers within the sender's window represent messages sent but not yet acknowledged. If the messages currently in the sender's window may ultimately be lost or damaged in transit, the sender must keep them for possible retransmission. A flow control scheme for point-to-point communication will as well regulate the flow of messages so that the number of 'unacknowledged' messages sent by a given process does not exceed the sender buffer's limits.

Flow control for multicast protocols have not received much attention in the literature. As discussed earlier, in multicast communication not only sent messages must be kept for possible retransmission but also received messages sent by all processes of a group. The message flow must then be controlled so that the number of such 'unstable' messages from different processes does not exceed a given limit on the buffer space of a process buffer. The importance of flow control as an implementation requirement for multicast protocols has been pointed out in [Amir92a]. The flow control scheme implemented in the Transis system is outlined in [Amir92a], and to our best knowledge is the only (very brief) published description of an implemented flow control scheme for multicast protocols. We will briefly compare the Transis flow control mechanism to our solution. In Transis, unstable messages are kept in a buffer and the flow control will work to avoid as much as possible the buffer overflow. Each process maintains a sliding window consisting of unstable messages (they are computed from the local process DAG's - see section 2.6.6). When the window exceeds a maximal size, the flow of messages is blocked. This will reduce the chances of buffer overflows in other processes. In Transis, acks (i.e. acknowledgments for received messages) are usually piggybacked on messages. However, if there is are messages to be transmitted or if the flow of messages is blocked, extra ack messages may be transmitted. Our flow control mechanism in contrast guarantees that no buffer overflows will occur. We guarantee that the buffer size will never exceed the stated bound even when processes are faulty. In our mechanism, senders may be temporarily blocked because of slow processes (or

process failures) but eventually they will be allowed to send new messages. Finally, our flow control mechanism does not exchange any extra (ack) messages in order to compute stability information. We achieve that by enforcing completeness and stability conditions on created Causal Blocks.

8.4 Conclusions

Flow control is an important aspect of protocol design. However, it has not received much attention in the literature for fault-tolerant multicast protocols. We have presented a flow control mechanism for a multicast protocol that ensures that a sender process does not cause buffers to overflow from a known limit at any of the functioning destination processes. This is achieved by piggybacking block completion and block stability information (two integer values) on top of normal messages. This permits a process to compute whether it is safe to transmit a message. Although the mechanism has been designed and implemented for a specific protocol, Newtop, the ideas presented can be adapted to any multicast protocol and will require the maintenance of a logical clock and an LRV vector for each group a process belongs to.

Chapter 9 - Conclusions

9.1 Synopsis

In this thesis, we have discussed the main issues concerning the design of fault-tolerant group communication protocols. We have assumed that processes can be geographically separated, communicating via long-haul network such as the internet. Asynchronous communication was therefore assumed where message transmission times cannot be accurately estimated and the network may well get partitioned. We have contributed in the area by presenting a comprehensive approach based on the concept of Causal Blocks for implementing fault-tolerant group communication protocols with different ordering requirements. Causal Blocks representation is based on the concept of logical clocks [Lamport78] and is a convenient way of maintaining and deducing ordering and reliability information between messages exchanged by processes of a group.

We have presented a symmetric total order protocol, called Newtop. Besides being simple to handle, even in the presence of overlapping groups, Newtop has the main advantage of the constant and low message space overhead. Symmetric total order protocols have also been presented in [Melliar-Smith90, Amir92a]. However, these protocols have not addressed the problem of multiple groups; further they assume in their system model a hardware broadcast facility on top of which the protocols are built (making them not as portable as Newtop). Asymmetric protocols with similar functionalities are described in [Chang84, Navarantnam88, Birman91b, Kaahoek91]. None of them however, addresses the group overlapping problem for total order delivery. In [Peterson89] is described a symmetric protocol where group overlapping is not addressed either. Protocols described in [Biman91b, Melliar-Smith90, Amir92a, Peterson89[1]] provide total order message delivery (for non-overlapping groups) but their protocols have been built on top of their respective causal order protocols and

[1] In fact, Psync provides a poit-to-point causal order protocol. However, a total order multicast protocol can be built on top of this basic primitive as discussed in [Peterson89].

require a larger message space overhead (the amount of ordering information added to application messages) when compared to Newtop. This is (mainly) due to the fact that our protocol has been built directly to provide total order (rather than being built on top of an existing causal order protocol). In [Birman91b] messages carry vector clocks. in [Peterson89, Amir92a] messages carry references to other messages directly related by the 'happened before' relation. In [Melliar-Smith90] messages carry positive and negative acknowledgements for other transmitted messages. In Newtop, in contrast, messages are timestamped with just a block-number. In order to assess delivery conditions, the protocols in [Melliar-Smith90, Amir92a, Peterson89] have to examine some 'stability' conditions in the structures they maintain for representing causal relationship between exchanged messages (negative and positive acks, the DAG graph, and the context graph, respectively). In Newtop, this is done just by keeping the minimum value of a vector of integers (the CBV vector) and by comparing this value with timestamps of received messages (integer values). Newtop is able to offer these advantages because it does not attempt to precisely represent the absence of causal relation among multicast as this is not essential for total order message delivery.

The protocol presented in [Garcia-Molina91] indeed addresses the group overlapping problem. However, all processes have to maintain the propagation graph that is built based on the group overlapping structure. Modification on the group memberships (due to process crashes or application related reasons) will lead to modification of the propagation graph, making it expensive to maintain for systems where groups change dynamically and frequently.

We have also presented three approaches for causal order message delivery in overlapping process groups. They represent different trade-offs between delivery delay and message space overhead costs. The Slow causal protocol produces the smallest timestamp (the message block-number) and the potentially longest delivery time since Causal Blocks for all the groups a process belongs to have to be complete before message delivery. The Fast causal protocol is based on the GLDV vector which is a precise representation of causal dependence between transmitted messages in a multi-group environment. As in CBCAST [Birman91b], the Fast causal protocol produces message delivery as early as possible. That is, the delivery of a message m is

only delayed if there is a message m', such that m' → m and m' has not been delivered yet. On the other hand, the Fast causal protocol imposes the highest message space overhead among the approaches presented (message overhead similar to CBCAST: an integer number per process belonging to all groups). The Relative causal protocol is a trade-off solution. It provides as early as possible delivery in a uni-group environment but some extra delay is possible in a multi-group environment due to the need of block-completion. Protocols presented in [Birman91b, Amir92a, Peterson89, Mostefaoui93] provide causal order delivery. However, overlapping groups are only addressed in [Birman91b, Mostefaoui93]. Our Relative Causal protocol presents a similar trade-off to the protocol presented in [Mostefaoui93]. In [Mostefaoui93] a message will carry an integer number per group involved in an overlapped structure. In our Relative causal protocol a message will carry an integer per group and also a vector of integers with the size of the group the message was sent to. Message delivery within a group will then be as early as possible (fast delivery) but Causal Blocks for other (possibly) overlapped groups have to be complete before message delivery can take place. Message delivery in [Mostefaoui93] will depend on the progress of the logical clocks corresponding to all groups a given process is a member of.

We have developed a failure suspector based on the Causal Blocks representation and local time-outs. Based on this failure suspector, we have deleloped a membership protocol that ensures that network partitions do not lead to processes forming inconsistent group membership views; further, message delivery is kept atomic with respect to view change installations. We have then developed a fault-tolerant total order protocol for overlapping process groups which works correctly despite process crashes and network partitions. To our best knowledge there is no other existing protocol fulfilling these ordering and reliability requirements all together.

The causal order protocols as well as the fault-tolerant mechanisms we have developed were designed to work in the context of Causal Blocks (i.e., using message block-numbers). Therefore, they can easily be integrated to provide causal ordering delivery in the presence of process crashes and network partitions.

We have described the implementation of the Newtop protocol over a set of networked UNIX sparc stations. We have run experiments to evaluate the performance of Newtop under varied group configurations, transmission rates, and time-silence timeouts, when processes crashes were not considered. The value set for the time-silence timeout is determinant on the number of null messages transmitted. When only one member of a group is transmitting messages, choosing a shorter time-silence timeout will on one hand cause a larger number of null messages be transmitted, but on the other hand, it will lead to a shorter message delay overhead and reduction in the number of unstable blocks produced. When all group members are actively transmitting messages at the same rate, varying the time-silence timeouts does not cause a great impact on the number of unstable blocks and the average delay overhead improved only slightly for shorter time-silence timeouts. So, choosing the appropriate time-silence value is a key point for tuning the behaviour of Newtop to specific application and system requirements such as local buffers (maximum number of unstable blocks), delay overhead, and consumption of transmission bandwidth (extra null messages transmitted).

We have developed a novel flow control mechanism for a multicast protocol that ensures that a sender process does not cause buffers to overflow from a known limit at any of the functioning destination processes. This is achieved by piggybacking block completion and block stability information (two integer values) on top of normal messages. This permits a process to compute whether it is safe to transmit a message. Although the flow control mechanism has been designed and implemented for Newtop, the ideas presented can be adapted to any multicast protocol and will require the maintenance of a logical clock and an LRV vector for each group a process belongs to.

Finally, all protocols and services such as membership, the flow control, *time-silence*, and the suspector have been developed based on the Causal Blocks representation and therefore they work in a integrated manner, without relying on any external service. This makes our work distinctive from existing works in the area.

9.2 Future Work

We have presented a membership protocol that can deal with process crashes and network partitioning. Process crashes can lead to situations where a group is sub-devided into sub-groups of functioning processes and our membership protocol was designed so that message delivery will be kept atomic with respect to view installations in each of the partitioned sub-groups. However, we have not addressed the problem of merging sub-groups holding the same group identifier, which could happen, for instance, after a partitioned network has been fixed. We are currently analysing the possibility of addressing the merging problem by taking the more general approach of group formation. In a group formation problem, a given process p_i knowing of the existence of a set g_n of processes, will make an attempt to form a new group (we assume that p_i is not a member of any g_x such that $V_{x,i} = g_n$). The processes involved in a given group formation process, can approve the attempt or reject it. There are a couple of questions to be investigated such as how many group formation attempts a given process is allowed to undertake and how a given group formation attempt can be distinguished from previous ones in an asynchronous environment. Once the group formation problem has been properly addressed, merging existing partitioned sub-groups can be dealt with in a straightforward way by making partitioned sub-group carrying out group formation attempts after events such as recovery from network partitioning. Notice that the problem of a process p joining a group g corresponds to merging g with the "1-member group" formed by p. Therefore, joining processes to existing groups can also be thought of as a particular case of the group formation problem.

The symmetric Newtop protocol we have developed provides a simple way of dealing with overlapping groups. As it is the case for any symmetric total order protocol, message delivery speed by Newtop will depend on the flow of messages from all the members of a group. When members are not active in sending messages, message delivery speed can slow down to the level of the most inactive process. In order to guarantee message delivery speed will not be affected by the inactivity of

some processes, we have used the time-silence mechanism that will force inactive members to transmit null messages. In an asymmetric protocol, where message delivery order is determined by a sequencer [Chang84], the activity of group members will not affect message delivery speed. We are presently investigating the use of sequencer processes to work on top of Newtop. Asymmetric solutions for total order delivery are difficult to handle because the choice of a sequencer will usually depend on the nature of the overlapped groups. For instance, in [Garcia-Millina91] nodes of the propagation graph will be common members of overlapped groups. It is possible to develop a Newtop asymmetric protocol where sequencer processes will timestamp messages with Causal Blocks numbers. Since messages are timestamped with increasing block-numbers (or logical clock values), we do not require that members of common groups will access the same sequencer, making simple to handle dynamic overlapping groups. We are also investigating the possibility of integrating symmetric and asymmetric multicasts so that a given process could send a message m to one group using the symmetric Newtop and send a message m' to another group using the asymmetric Newtop. Thus, applications will be able to take advantage of this flexibility to achieve best performance according to distinct environments. For instance, symmetric Newtop version will be more suitable to environments where processes are actively transmitting new messages, whereas asymmetric Newtop version will be more efficient in environments where only a few group members are active in sending messages.

Our reliable transport layer has been built on top of TCP/IP which is a protocol optimised for stream communication. We expect that by implementing Newtop on top of an unreliable datagram suite such as UDP/IP we will get a better performance (in section 3.7, we have explained how missing messages can be represented in the Block Matrix). The implementation of the transport multicast layer using UDP/IP and the analysis of this new implementation on the performance of Newtop is left for the future. Similarly, performance analysis of Newtop in a multi-group environment, and

implementation of the Relative causal order protocol and its integration with the membership service is also left for the future.

References

[Amir92a]
Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki, "Transis: A Communication Sub-System for High Availability", 22nd International Symposium on Fault-Tolerant Computing (FTCS-22nd), Boston, July 1992.

[Amir92b]
Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki, "Membership Algorithm for Multicast Communication Groups", Proc. of the 6th International Workshop on Distributed Algorithm, pp 292-312, November 1992.

[Babaoglu88]
O. Babaoglu, P. Stephenson, and R. Drummond, "Reliable broadcasts and communication models: tradeoffs and lower bounds". Distributed Computing (1988) 2: 177-189, Spring Verlag.

[Birman87]
Birman, K. and Joseph, T. "Reliable Communication in the Presence of Failures", ACM Transactions on Computers Systems 5, 1 (Feb. 1987)

[Birman89]
Birman, K. and Joseph, T., "Exploiting Replication in Distributed Systems", In Sape Mullender, editor, Distributed Systems, New York, 1989, ACM Press, Addison-Wesley.

[Birman91a]
Kenneth P. Birman, "The Process Group Approach to Reliable Distributed Computing", Technical Report TR91-1216, Department of Computer Science, Cornell University, July, 1991.

[Birman91b]
Birman, K., Shiper A., and Stephenson, "Lightweight Causal and Atomic Group Multicast", ACM Transactions On Computer Systems, Vol. 9, No 3, August 1991, pp. 272-314.

[Birman91c]
Kenneth P. Birman, "Design Alternatives for Process Group Membership and Multicast", Technical Report TR91-1257, Department of Computer Science, Cornell University, December, 1991.

[Chandra91]
T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Asynchronous Systems", Proceedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing, pages 325-340. ACM, August 1991.

[Chandy85]
K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States in Distributed Systems", ACM Transactions on Computer Systems, Vol. 3, No. 1, pp. 63-75, February, 1985.

[Chang84]
Chang, J. and Maxemchuk, N. F. "Reliable Broadcast Protocols". ACM Transactions on Computer Systems, Vol. 2, No 3, August 1984, Pages 251-273.

[Charron91]
B. Charron-Bost B., "Concerning the size of Logical Clocks in Distributed Systems", Inf. Proc. Letters, Vol. 39, (1991), pp. 11-16.

[Cheriton85]
D. R. Cheriton and W. Zwaenepol, "Distributed Process Groups in the V Kernel", ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985.

[Cristian90]
Flaviu Cristian, Danny Dolev, Ray Strong, Houtan Aghili, "Atomic Broadcast in a Real-Time Environment", Lecture Notes in Computer Science, No. 448, pp. 51-71.

[Cristian91]
Cristian, F., "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems", Distributed Computing, Vol. 4, No. 4, April 1991, pp. 175-187.

[Dolev93]
Danny Dolev, Shlomo Kramer, Dalia Malki, "Early Delivery Totally Ordered Multicast in Asynchronous Systems", FTCS-23, Toulouse, France, June, 1993.

[Ezhilchelvan93]
Paul Ezhilchelvan and Santosh K. Shrivastava, "rel/REL: A Family of Reliable Multicast Protocols for Distributed Real-Time Systems", Technical Report 461, Department of Computing Science , Newcastle University, 1993.

[Fidge91]
C. J Fidge, "Logical time in distributed computing systems, IEEE Computer, Vol. 24,8 (1991).

[Fischer83]
M. Fischer, "The Consensus Problem in Unreliable Distributed Systems". Proceedings of the Internatiaonal Conference on Foundations of Computing Theory, Sweden, 1983.

[Fischer85]
M. Fischer, N. Lynch, and M. Peterson, "Impossibility of Distributed Consensus with One Faulty Process", J. ACM, 32, April 1985, pp 374-382.

[Garcia-Molina91]
Hector Garcia-Molina, "Ordered and Reliable Multicast Communication", ACM Transations on Computer Systems, Vol. 9, No. 3, August, 1991, pages 242-271.

[Kaashoek91]
M. Frans Kaashoek and Andrew S. Tanenboum, "Group Communication in the Ameoba Distributed Operating System", Proc. Eleventh International Conference on Distributed Computing Systems, Arlington, TX, May 1991.

[Lamport78]
Lamport, L., "Time, clocks, and ordering of events in a distributed system", Commun. ACM, 21, 7 (July 1978), pp. 558-565.

[Lamport82]
L. Lamport, R.. Shostak, and M.Pease, "The Byzantine Generals Problem". ACM Transactions on Programming Languages and Systems, vol. 4, n0. 3, pp. 382-401, July 1982.

[Hadzilacos93]
Vassos Hadzilacos and Tam Toueg, "Fault-Tolerant Broadcast and Related Problems". In Distributed Systems (Second Edition), edited by Sape J. Mullender, ACM Press, New York, 1993.

[Macêdo93a]
Raimundo A. Macêdo, Paul Ezhilchelvan, Santosh K. Shrivastava, "Modelling Group Communication using Causal Blocks", 5th European Workshop on Dependable Computing, Lisbon, February, 1993.

[Macêdo93b]
Raimundo A. Macêdo, Paul Ezhilchelvan, Santosh K. Shrivastava, "Newtop: a Total Order Multicast Protocol Using Causal Blocks", Broadcast deliverable report, Volume I, First open Broadcast workshop, Newcastle, october, 1993.

[Mattern89]
F. Mattern, "Time and global states in distributed systems", In Proc. of the International Workshopom Parallel and Distributed Algorithms, North-Holland, Amsterdam, 1989.

[Melliar-Smith90]
M. P. Melliar-Smith, L. E. Moser, and V. Agarwala, "Broadcast Protocols for Distributed Systems", IEEE Transactions on Paralell and Distributed Systems, Vol. 1, No. 1, January, 1990.

[Melliar-Smith91]
M. P. Melliar-Smith, L. E. Moser, and V. Agarwala, "Membership Algorithms for Asynchronous DIstributed Systems", Proc. of the 12th International Conf. on Distributed Comp. Systems, pp 480-488, May 1991.

[Mishra91]
S. Mishra, L. Peterson, and R, Schlichting, "A Membership Protocol Based on Partial Order", Proc. IFIP Conf. on Dependable Computing For Critical Applications, Tuscon, Feb. 1991, pp 137-145.

[Mishra93]
Mishra, S., Peterson L., and Schlichting, R., "Consul: a Communications Substrate for Fault-Tolerant Distributed Programs", Distributed Systems Engineering, 1 (1993), pp. 87-103.

[Mostefaoui93]
Achour Mostefaoui and Michel Raynal, "Causal Multicasts in Overlapping Groups: Towards a Low Cost Approach", In Proc. of the 4th IEEE Int. Conference on Future Trends of Distributed Systems, pp. 136-142, Lisboa, September 1993.

[Navaratnam88]
S. Navaratnam, S. Chanson, and G. Neufeld, "Reliable Group Communication in Distributed Systems", Proc. 8th Int. Conf. on Dist. Comp. Sytems, San Jose, CA, pp. 439-446 (June 1988)

[Olsen91]
Michael H. Olsen, Ed Oskiewicz, and John P. Warne, "A Model for Interface Groups", Proceedings of the Tenth Symposuim on Reliable Didtributed Systems, Pisa, Italy, October, 1991.

[Peterson89]
L. L. Peterson, N. Bucholz, and R Schlichting, "Preserving and using context information in interprocess communication", ACM Transactions on Computer Systems, Vol. 7, No 3, August 1989, pp. 217-246.

[Raynal91]
Michel Raynal, André Schiper, and Sam Toueg, "The causal ordering abstraction and a simple way to implement it", Information Processing Letters, Vol. 38, pp. 343-350, 1991.

[Raynal92]
Michel Raynal, "About Logical Clocks for Distributed Systems", Operating Systems Review, SIGOPS, vol. 26, Number 1, January, 1992.

[Ricciardi91]
A. M. Ricciardi and K.Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments", Proc. of Annual ACM symposium on PoDC, pp. 341-352, August 1991.

[Ricciardi92]
Aleta Marie Ricciardi, "The Group Membership Problem in Asynchronous Systems", Ph.D. dissertation, Cornell University, 1992.

[Rodrigues91]
L. Rodrigues and P. Verissimo, "xAMp: A multi-primitive Group Communication Service", Proc. of the eleventh Symposium on Reliable Distributed Systems, Houston, TX, 1991.

[Schiper89]
A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering", Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes on Computer Science 392, Springer-Verlag, New York, 1989, pp. 219-232.

[Schiper93a]
A. Schiper and A. Sandoz, "Understanding the power of the virtually-synchronous model", 5th European Workshop on Dependable Computing, Lisbon, February, 1993.

[Schiper93b]
A. Schiper and A. Sandoz, "Uniform Reliable Multicast in a Virtually Synchronous Enviroment", IEEE proceedings of the 13th Int. Conf. On Distributed Computing Systems (ICDCS-93), Pittsburgh, May 25-28, 19993.

[Schiper93c]
A. Schiper and A. Ricciardi, "Virtually Synchronous Communication Based on Weak Failure Suspector", IEEE Proc. of the 23rd Annual Int. Symp. on Fault-Tolerant Computing (FTCS-23), Toulose, June/1993.

[Schmuck88]
Frank Schuck, "The use of Efficient Broadcast Primitives in Asynchrounous Distributed Systems", Ph.D. dissertation, Cornell University, 1988.

[Stevens90]
W. R. Stevens, "Unix Network Programming", Prentice-Hall, Englewood Clifes, N. J., 1990.

[Strong83]
R. Strong and D. Dolev, "Byzantine agreement", Digest of Papers, Spring Compcon, IEEE Computer Society Press, 1983.

[Tamenbaum81]
A. S. Tanenbaum, "Computer Networks", Prentice-Hall, 1981.

[Verissímo93]
Paulo Verissímo, Luís Rodrigues, and Werner Vogels, "Group Orientation: a Paradigm for Modern Distributed Systems", BROADCAST Project deliverable report, vol I, October 1993.