Fast Ordered Multicasts

Patrick Stephenson
Ph.D. Thesis

TR 91-1194
February 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

FAST ORDERED MULTICASTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Patrick Stephenson

May 1991

FAST ORDERED MULTICASTS

Patrick Stephenson, Ph.D.

Cornell University 1991

In this thesis, we present new protocols that provide reliable ordered multicasts to multiple overlapping process groups in the presence of failures. Our protocols provide two kinds of message delivery ordering – causal ordering and total ordering. Message delivery is also ordered with respect to the observation of group membership changes, a property known as *virtual synchrony.* Initially we examine solutions for the case of a single process group, and subsequently extend our solutions to encompass multiple overlapping process groups. In comparison with previous protocols for these problems, our protocols are cheaper and scale up better. An initial implementation of our protocols as part of the ISIS toolkit has produced encouraging performance results.

Biographical Sketch

Pat Stephenson was born in Dublin, Ireland, in 1961. He attended St. Mary's College, Rathmines, graduating with his Leaving Certificate in 1979. He attended Trinity College, Dublin, graduating in 1983 with a Foundation Scholarship, a B.A. in Computer Science, and a Gold Medal. He then spent a year working for Compucorp, Inc., of Los Angeles. He was lucky enough to spend part of this year in Cork, Ireland. He came to Cornell in 1984, and received his M.S. in Computer Science in January 1987. In July 1989, he married Jeannette Clare Poleman.

To Clare

Acknowledgements

I am indebted to many people for the assistance they have given me. First and foremost, I must thank my wife, Clare, for all her love, encouragement, faith, support and sharing. Her presence in my life has made it infinitely more meaningful. This thesis could be dedicated to no-one else.

My parents, Pat and Niamh, taught me much that is important in life. In particular, they instilled in me an appreciation for knowledge, and for the value of a balanced life. Throughout my life, they have always been at my side with advice and encouragement. I am deeply grateful to them.

Cyril Byrne both taught me to appreciate mathematics and suggested that I might like to study computer science. His incisive style of intellectual inquiry has always been a model to me.

Throughout my time at Cornell, the enthusiasm, encouragement and technical advice of Ken Birman has been invaluable. Without him, this thesis would never have been completed. He has been the perfect advisor, in both the professional and personal sense.

Keith Marzullo gave good counsel when it was sorely needed, and also served on my thesis committee. Özalp Babaoğlu introduced me to good research before he left Ithaca for warmer climes. Much later, he introduced me to good Italian food. To both of them, my gratitude.

The Computer Science Department at Cornell has been a wonderful environment in which to work. The faculty and students combine to make Ithaca one of the finest places in the world to study computer science. I am particularly indebted to Tushar Chandra, Jacob Aizikowitz, Robert Cooper, Ajei Gopal, Gil Neiger and Amitabh Shah for many valuable technical discussions. I would also like to thank Prakash Panangaden and Keith Dennis for serving on my thesis committee.

I have been lucky enough over the years to live with some of the most tolerant roommates to be found in Ithaca, or, indeed, anywhere. Without them, the vicissitudes of graduate student life would have been much harder to bear. They include Chris Bischof, Kieran Herley, James Stewart and Kjartan Stefansson. Through the last few years, my friendships with Mike O'Donnell, Suresh Goyal and Amitabh Shah have been invaluable. To them all, my heartfelt thanks — especially for the beer.

I am grateful for the financial support of General Electric, IBM, and the ISIS project. The ISIS project is supported in part by DARPA under DARPA/NASA subcontract NAG-2-593 administered by NASA Ames Research Center. The views, opinions and findings in this thesis are entirely my own.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed computer systems are becoming ubiquitous. Despite this, remarkably little of the application code that runs on them takes advantage of distribution in any but the most trivial fashion. This is because the design and implementation of distributed programs is hard; for two major reasons. First of all, in a distributed system, there are many loosely-coupled parallel threads of control. Communication between these threads may be asynchronous and unreliable. Since the executions of each individual process in the system may be interleaved in an arbitrary fashion, the number of possible states of the system is exponentially larger than the number of states of any individual process.

Secondly, a distributed system, unlike a single computer, is subject to *partial* failures. These occur when some but not all of the processors comprising the system fail. Applications running on distributed systems are typically expected to survive such partial failures. This fault-tolerance requirement greatly increases the number of possible states of the system. It is this requirement (not communication latency) that distinguishes distributed systems from parallel systems; a parallel system is typically assumed to fail as a whole.

These two factors combine to make distributed systems complex to design, reason about and understand. Much of computer science can be understood as an effort to contain, manage, and reduce the almost limitless complexity of computational processes. This thesis concerns an effort to reduce the complexity of programming distributed computer systems.

1.1 Process Groups

In order to manage distributed systems, many researchers have turned to the abstraction of *process groups*, for example [CZ85,OSS80]. A process group is simply a collection of processes cooperating in order to achieve a common goal. The group has a name, which is independent of the location in the system of any individual member process. Typically, processes can send messages to the process group as a whole (*multicast* messages), as well as to individual members of the group *point-to-point messages.* Whether processes outside the process group communicate with processes inside the process group via multicast or point-to-point messages depends on the particular system under consideration. A process group can be viewed as providing a service to the distributed system as a whole. This is of some assistance in the management of complexity — the process group will hide from its clients the fact that it is distributed, fault-tolerant, and so on.

However, the execution of such a system can still be rather complex and difficult to manage. An example is shown in Figure 1.1. In this example, processes P2 through P4 make up a process group. Messages are being sent and delivered using standard IPC mechanisms, which provide point-to-point reliability and little else. Processes P1 and P5 multicast messages B1 and B2 respectively to the process group around the same time. Note that some members of the group receive B1 before they receive B2; other members of the group receive the two messages in the opposite order. When dealing with these messages, each process must take into consideration the fact that other processes may have observed message deliveries in a different order. Messages B3 and B4 provide a further example; even though process P2, the sender of B4, had already received B3, (and was perhaps sendi-

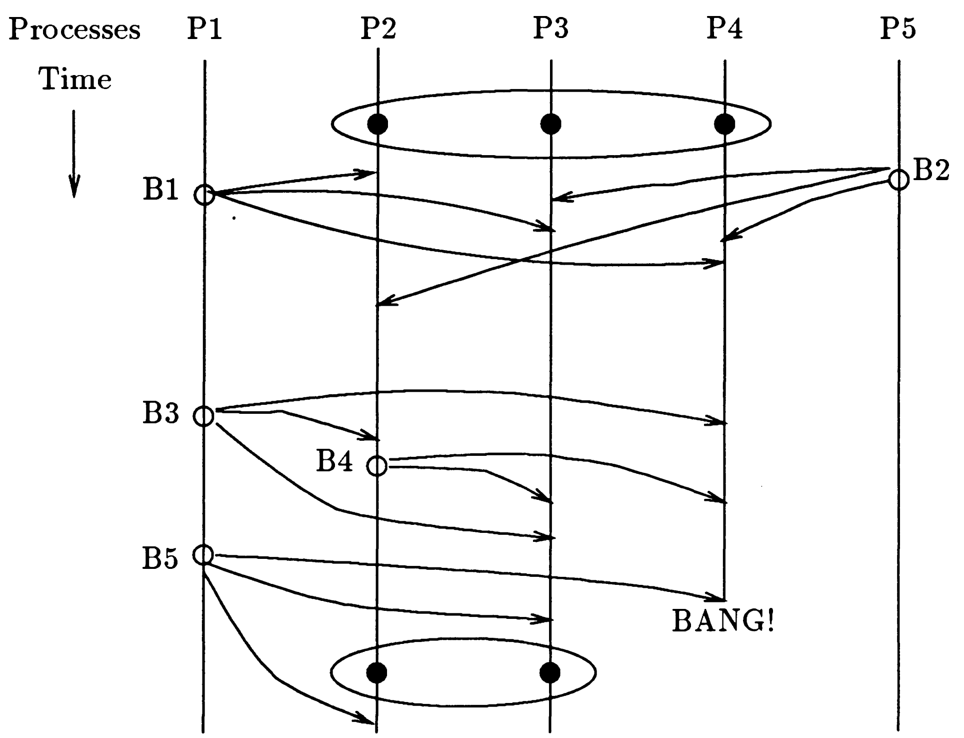

Figure 1.1: An asynchronous system execution

B4 *in response* to B3) process P3 receives B3 before it receives B4. When failures occur, message ordering become even more confused; in this example, P2 observes the failure of P4 before it receives B5, whereas P3 receives B5 and then observes the failure of P4. The programmer who designs these processes will clearly have a lot of cases to handle!

1.2 Virtual Synchrony and Multicast Ordering

The ISIS distributed programming toolkit [BJ87a,BJ87b] is an effort to overcome the problems outlined in the previous section. ISIS attempts to provide the programmer with a *virtually synchronous* programming environment. In such an envi-

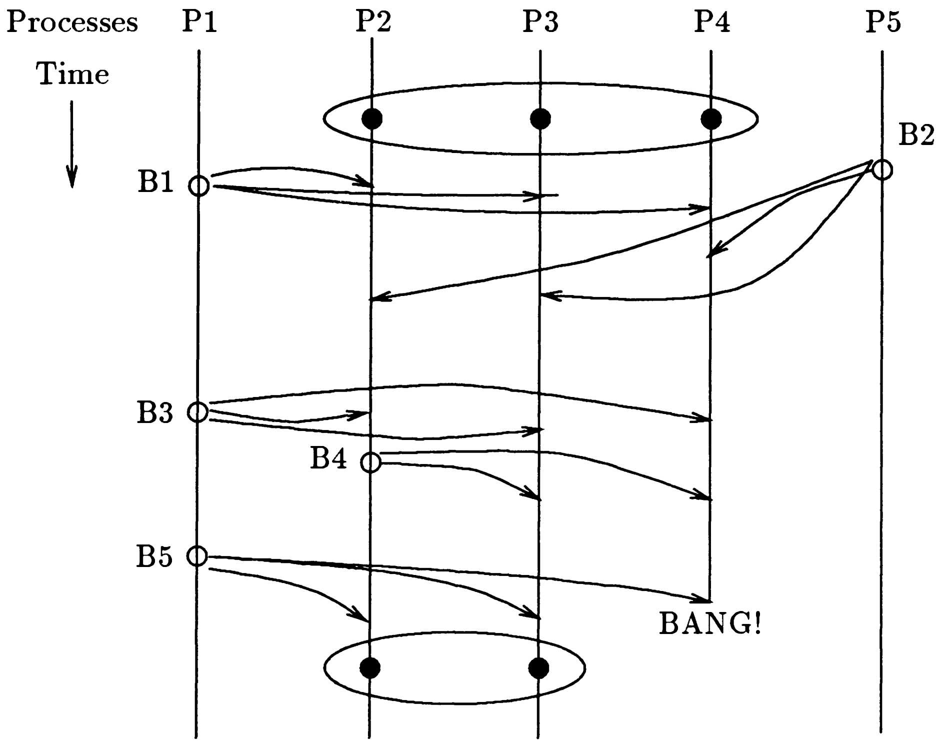

Figure 1.2: A virtually synchronous system execution

ronment, it will appear to any process using the system that all processes observe the *same* events in the *same* order. This applies both to message deliveries and group membership changes. In particular, messages are delivered atomically with respect to process failure. This observed common ordering considerably simplifies the application-level design of each process in the group, since each process can *infer* a considerable amount of information about the state of other processes in the group simply from the message orderings it has observed.

Consider Figure 1.2, which shows the execution of the previous example in a virtually synchronous environment. Now, multicast B1 is received before multicast B2 at each process in the group; B4 is received after B3 at P3 and P4; and B5 is received at P2 and P3 before the failure of P4 is observed. Note that the relationship between B1 and B2 is not the same as the relationship between B3 and B4. Since P2 received B3 before it sent B4, we say that B3 and B4 are causally related. No such relationship exists between B1 and B2. The ISIS toolkit provides a primitive that can order unrelated messages like B1 and B2; this primitive is called ABCAST. It also provides a primitive that maintains causal orderings; this primitive is called CBCAST. We will see in Chapter 3 that it is cheaper and easier to impose ordering on messages that are causally related, such as B3 and B4. In other words, CBCAST is cheaper than ABCAST. Schmuck [Sch88] has described a large class of distributed algorithms that can be implemented using CBCAST. Currently, CBCAST accounts for the majority of communication in the system. The protocols developed in this thesis are built in whole or in part on CBCAST; clearly, the correctness and performance of CBCAST will be vital.

1.3 Types of process groups

The concept of virtual synchrony has proved to be enourmously useful, and ISIS has been distributed to several hundred sites, and is presently used in many settings such as brokerage and banking applications, wide-area seismic data collection and analysis, factory floor automation, distributed simulation, scientific computing, high-availability file management, reactive control, education, and research [BC90]. In examinging these settings, we have seen four basic styles of group usage, illustrated in Figure 1.3. Many ISIS applications employ more than one of these structures, employing overlapping groups when mixed functionality is desired. The simplest of these is denoted the *peer group*. In a peer group, processes cooperate as equals in order to get a task done. They may manage replicated data, subdivide tasks, monitor one another's status, or otherwise engage in a closely coordinated distributed action. Another common structure is the *client/server group*. Here, a peer group of processes act as servers on behalf of a potentially large set of clients.

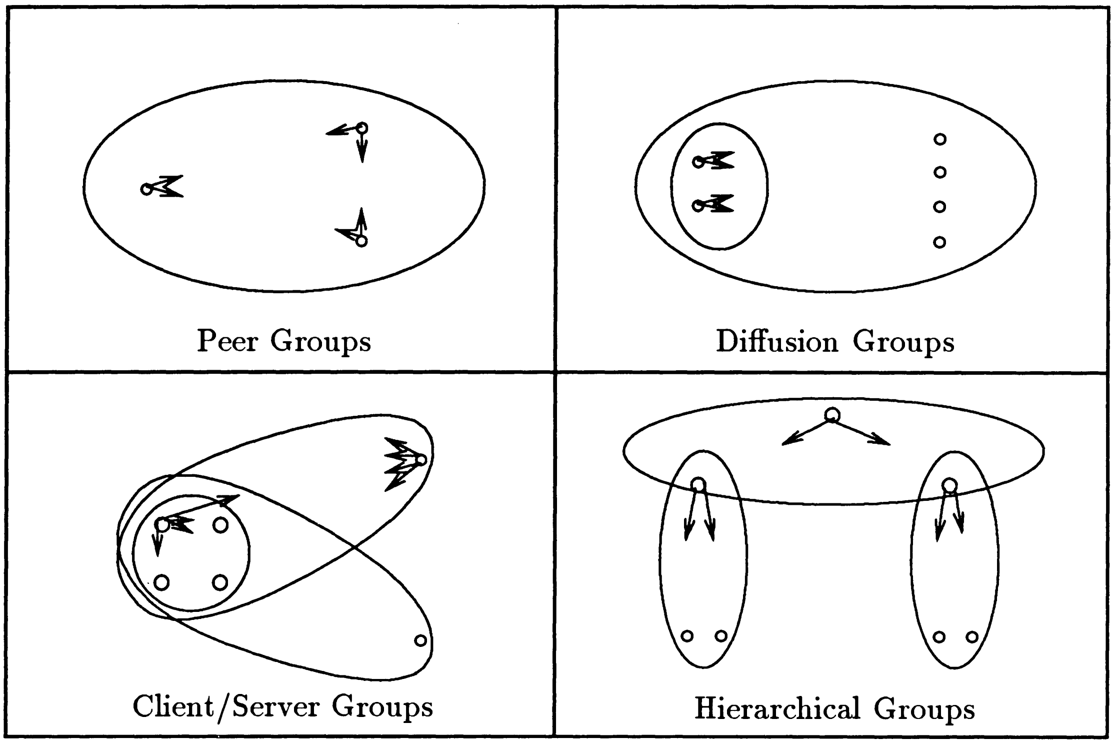

Figure 1.3: Types of process groups

Clients interact with the servers in a request/reply style, either by picking a favorite server and issuing RPC calls to it, or by multicasting to the whole server group. In the later case, servers will often multicast their replies both to the appropriate client and to one another. A *diffusion group* is a type of client-server group in which the servers multicast messages to the full set of servers and clients. Clients are passive and simply receive messages. Diffusion groups arise in any application that multicasts information to large number of sites, for example on a brokerage trading floor.

Finally, *hierarchical* group structures arise when large server groups are needed in a distributed system [CB89,GMS89]. Hierarchical groups are tree-structured sets of groups. Application programs interact with such groups without knowledge of their internal organization. A *root* group maps the initial connection request to an appropriate subgroup, which operates on behalf of the application. Data is partitioned among the subgroups, and although a large-group communication mechanism is available, it is rarely needed.

1.4 Thesis contributions

This thesis describes a new suite of protocols in support of the ISIS computation model. The original ISIS protocols were costly in part for structural reasons and in part because of the protocol used [BJ87b]. The implementation was within a protocol server, hence all communication was via an indirect path. Independent of the cost of the protocols themselves, this indirection was expensive. Furthermore, the protocol server proved difficult to scale, limiting the initial versions of ISIS to networks of a few hundred nodes. With respect to the protocol used, our initial implementation favored generality over specialization, permitting extremely flexible destination addressing. It used a *piggybacking* algorithm that achieved the CBCAST ordering property but required periodic garbage collection. The case for flexibility seems weaker today. Experience with ISIS has left us with substantial insight into how the system is used, permitting us to focus on core functionality.

In our new protocols, each process communicates directly with other processes in groups of which it is a member, and with no other processes. In addition, the overhead associated with piggybacked information is normally bounded in proportion to the size of the process groups to which the sender of a message belongs. Our protocol architecture permits application builders to define new transport protocols, perhaps to take advantage of special hardware. As a consequence, the new protocol suite can support highly concurrent applications and scale to systems with large numbers of potentially overlapping process groups.

Although addressing is somewhat less general than the earlier solution, the new protocols are able to support the ISIS toolkit and all ISIS applications with which we are familiar. The benefit of this reduction in generality has been a substantial increase in the performance and scalability of our system. In fact, the new protocol suite has no evident limits to the scale of system it could support.

1.5 Thesis outline

The next chapter gives our system model, which describes the environment within which we will solve our problems. Chapter 3 gives a series of protocols for providing virtual synchrony and multicast ordering within a *single* process group. Chapter 4 gives a general method for extending the single-group protocols of Chapter 3 to work in settings with multiple, dynamically changing process groups. This algorithm is general enough to extend to other single-group protocols that have appeared in the literature. It can also be applied to solve several specific problems, such as point-to-point messages and large client-server groups. Chapter 5 gives a large class of optimizations that can be applied to the algorithm of Chapter 4, in cases where the group overlap structure is known or can be inferred. Chapter 6 discusses the performance of our protocols and gives some preliminary performance figures. Finally, Chapter 7 gives some pointers for further work.

Chapter 2

System Model

2.1 Introduction

In this chapter, we introduce and justify our system model. The model will be used to formally specify both the problems we are attempting to solve, and the facilities the system provides that can be used to solve these problems. Section 2.2 discusses the basic elements of our system, processes and groups. Section 2.3 formally defines the basic primitives upon which we will be building our protocols — a group membership service, discussed in subsection 2.3.1, and a message transport service, discussed in subsection 2.3.2. Section 2.4 gives the basic events that may occur in the system and explains how they are related to each other. Section 2.5 specifies the desired behaviour of the protocols that we are considering in terms of these basic events, and Section 2.6 concludes the chapter.

2.2 Processes and Groups

The system is composed of n processes $P = \{p_1, p_2, ..., p_n\}$ with disjoint memory spaces. We will initially assume that this set is static and known in advance; in Chapters 4 and 5 we relax this assumption, and make use of of the group membership service described in Section 2.3.1. Processes fail by crashing detectably (a *fail-stop* assumption; see [Sch84]); failure notification is provided via the group membership service discussed below.

In many situations processes will need to cooperate, as discussed in Chapter 1. For this purpose, they form *process groups.* Each such group has a name and a set of member processes; members join and leave dynamically. Additionally, when a member fails, it will depart from all process groups to which it belongs. The members of a process group need not execute the same code, nor is there any limit on the number of groups to which a process may belong. The set of process groups is denoted by $G = \{g_1, g_2 ...\}$. In some settings, the number of groups will be large and processes will belong to several groups.

2.3 Underlying primitives

2.3.1 Group Membership

Our system model is unusual in assuming an external service that implements the process group membership abstraction. Such a service can easily be provided by ISIS; in fact, it is one of the lowest layers of the current system. Others have also recognized the importance of this service in fault-tolerant distributed systems and designed algorithms that implement it; see [Cri90,RB90,RB91,MPS90]. The interface from a process to the membership service will not concern us here, but the manner in which the service communicates to a process is highly relevant. A *view* of a process group is a list of its members. A *view sequence* for g is a list $view_0(g), view_1(g), ..., view_n(g)$, where

1. $view_0(g) = \emptyset$.
2. $\forall i : view_i(g) \subseteq P$, where P is the set of all processes in the system.
3. We assume for simplicity that $view_i(g)$ and $view_{i+1}(g)$ differ by the addition or subtraction of exactly one process. The protocols developed in this thesis can can easily be generalized to operate where this assumption does not hold.

The membership service computes new group views in response to failures or process join/leave requests and delivers them to the members of the groups involved. Each member of a group g will receive exactly the sequence of views described by the view sequence of group g. If a process fails, every group of which it was a member will eventually receive a view which does not contain that process. Processes may join and leave groups by communicating with the membership service; the details of this interface will be discussed in Section 3.5. The latest view of a group g that has been received at a process p is called p's *current* view of g. Processes learn of the failure of other group members *only* through the view mechanism, never through any sort of direct observation. Once a process fails, it appears to remain failed forever — recovering processes are simply new members, and do not retain state from their previous incarnation.

The membership service for each group is independent of the membership service for any other group; as a consequence, if a set of processes are members of the same groups, the interleaving of group view deliveries may vary at each processor. For example, suppose that p_1, p_2 and p_3 are members of both g_1 and g_2. If p_3 fails, new views for both g_1 and g_2 will be delivered at both p_1 and p_2. In this situation, it is possible that p_1 will receive the new view for g_1 before receiving the new view for g_2, and p_2 will receive the new view for g_2 before receiving the new view for g_1.

2.3.2 Message Transport

We assume that direct communication between non-failed processes is always possible; the software implementing this abstraction is called the *message transport* layer. Within our protocols, processes always communicate by using point-to-point or multicast messages; the latter may be transmitted using multiple point-to-point messages if no more efficient alternative is available. Multicast messages are always sent to the entire current membership of a single group. If the sender fails while sending a multicast message, it may only be received by some subset of the processes to which it was sent. The transport communication primitives must provide lossless, uncorrupted, first-in-first-out (FIFO) message delivery. They should also be able to indicate which messages previously passed to them have in fact been received by their intended recipient processes. Since the transport primitives must maintain this information, providing it to upper layers should not be expensive.

2.4 Primitive Event Orderings

The execution of a process is a partially ordered sequence of *events*, each corresponding to the execution of an action that is indivisible with respect to failure of that process. An acyclic event order, denoted $\xrightarrow{p}$ reflects the dependence of events occurring at process p upon one another. The event $send_p(m, g)$ denotes the transmission of m by process p to process group g. The message m will be sent to all the processes in the current view of g. The reception of a such a message at a process p' that is in the current view of g is denoted by $rcv_{p'}(m, g)$. The group and process names will be omitted when the context is unambiguous. We will assume that *send* puts messages into all communication channels in a single action that might be interrupted by failure, but not by other *send* or *rcv* actions. A process can also send unicast messages, i.e. messages to only one other process. We denote the transmission of such a message from p to p' with the events $send_p(m, p')$ and $rcv_{p'}(m, p)$, where p and p' are members of P. We denote by $dests(m)$ the set of processes to which m was sent - either a single process or the current membership of some process group. The event $fail_p$ denotes the failure of a process p, and is the last event at any process in which it occurs. Every message m is received by each process in $dests(m)$ that does not fail. In addition, messages are received in the order in which they were sent. If a process p fails, then the group membership service will eventually notify every other process in each group g of which it was a member by delivering a new view $view(g)$, where $p \notin view(g)$, to the other members of g. We denote by $rcv_p(view_i(g))$ the event by which a process p belonging to g "learns" of $view_i(g)$. This event occurs at all processes that are members of $view_i(g)$, unless they do not fail.

We distinguish the event of *receiving* a message at a process from the event of *delivery*, since this allows us to model protocols that delay message delivery until some condition is satisfied. The delivery of new group views can also be delayed or omitted, in order to ensure *virtual synchrony*. The delivery event is denoted $deliver(m)$ where $rcv(m) \xrightarrow{\text{p}} deliver(m)$.

2.5 Ordered Multicast Specifications

The multicast protocols that interest us here provide delivery ordering guarantees. As in [Lam78], we define the potential causality relation for the system, $\rightarrow$, as the transitive closure of the relation defined as follows:

1. If $\exists p: \; e \xrightarrow{\text{p}} e'$, then $e \rightarrow e'$
2. $\forall m: \; send(m) \rightarrow deliver(m)$

Our first type of ordered multicast, CBCAST, satisfies a causal delivery property:

$$send(m) \rightarrow send(m') \Rightarrow$$
$$\forall p \in dests(m) \cap dests(m'): \; deliver(m) \xrightarrow{\text{p}} deliver(m'). \tag{2.1}$$

If two CBCAST messages are concurrent, the protocol places no constraints on their delivery ordering at overlapping destinations. If a process p multicasts a CBCAST m to a set of processes, all messages that have been delivered to p before it sent m will also be delivered at all the recipients of m before m itself is delivered. Intuitively, m will be delivered in the same context in which it was sent.

We now consider ways to extend the CBCAST ordering of causally related messages to orderings of messages that are not causally related to each other. There are several different methods of defining totally ordered multicast delivery in a system with multiple overlapping process groups. These vary both in the semantics they provide and in cost of implementation. These semantics are discussed in detail in [GT91]; we repeat their definitions here. The cheapest ordering to implement simply provides a total ordering over messages multicast to the *same* process group:

$$send(m) \rightarrow send(m') \Rightarrow$$
$$\forall p \in dests(m) \cap dests(m') : \; deliver(m) \xrightarrow{p} deliver(m'). \tag{2.2}$$
$$\exists m, m', p \in g : deliver_p(m, g) \xrightarrow{p} deliver_p(m', g) \Rightarrow$$
$$\forall q \in g : deliver_q(m, g) \xrightarrow{q} deliver_q(m, g) \tag{2.3}$$

Equation 2.2 is the same as equation 2.1, showing that our total ordering is an extension of causality. Equation 2.3 states that messages sent in the same group will be delivered in the same order at all processes that are members of that group. We will refer to this total ordering condition as *local* total ordering. A multicast that provides local total ordering is our second multicast primitive – ABCAST.

Now, if two processes are both members of the same two groups, and an ABCAST message is multicast in each group, the local total ordering condition does *not* specify that each of the processes should receive the two messages in the same order. In order to specify this, we can strengthen condition 2.3 above by adding:

$$\exists m, m', p \in g, g' : deliver_p(m) \xrightarrow{p} deliver_p(m') \Rightarrow$$
$$\forall q \in g, g' : deliver_q(m) \xrightarrow{q} deliver_q(m) \tag{2.4}$$

Conditions 2.3 and 2.4 together define *pairwise* total ordering. However, even pairwise total ordering does not define a total order over all messages in the system. Consider the scenario illustrated in Figure 2.1, where three processes are each members of two groups, and an ABCAST message is multicast in each group. It is not possible to give an overall global order of message delivery, since the union of the delivery orders at the three processes forms a cycle. If we wish to avoid

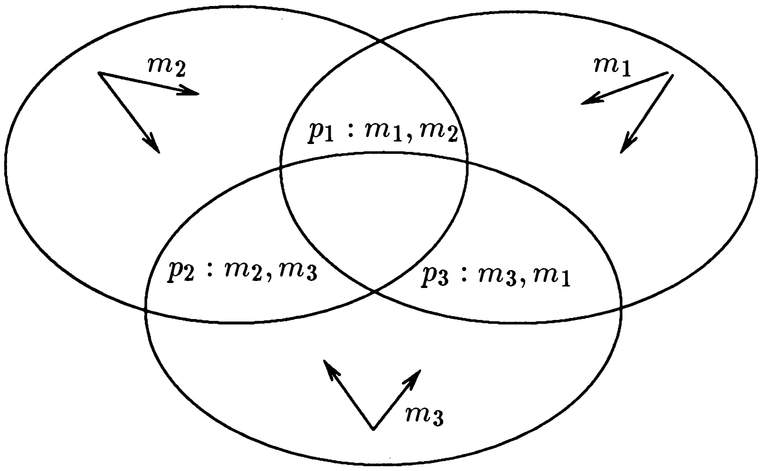

Figure 2.1: Inconsistent ABCAST delivery in multiple process groups

this situation, we must strengthen the previous semantics. We define the "must precede" relation, $\hookrightarrow$, over totally ordered multicasts as the transitive closure of the following relation:

$$\forall m, m' : \exists p : deliver_p(m) \xrightarrow{\mathrm{p}} deliver_p(m') \Rightarrow m \hookrightarrow m'$$

We can now define *global* total ordering:

$$\begin{aligned}&\forall m, m' : m \hookrightarrow m' \Rightarrow \\ &\qquad \forall p \in dests(m) \cap dests(m') : deliver_p(m) \xrightarrow{\mathrm{p}} deliver_p(m')\end{aligned} \tag{2.5}$$

This is known as the *global* total ordering condition.

Because an ABCAST protocol orders logically concurrent events, it will be more costly than CBCAST, requiring synchronous solutions where a CBCAST protocol admits efficient asynchronous solutions. Birman and Joseph [BJ89] and Schmuck [Sch88] have exhibited a large class of algorithms that can be implemented using CBCAST. Moreover, Schmuck has shown that in many settings algorithms specified in terms of ABCAST can be modified to use CBCAST without compromising correctness.

2.5.1 Virtual Synchrony

Both multicasts must satisfy the virtual synchrony property, introduced in Chapter 1:

$$\exists i, j, p : deliver_p(view_i(g)) \xrightarrow{p} deliver_p(m) \xrightarrow{p} deliver_p(view_j(g)) \Rightarrow$$
$$\forall p' \in view_i(g) : deliver_{p'}(view_i(g)) \xrightarrow{p'} deliver_{p'}(m) \xrightarrow{p'} deliver_{p'}(view_j(g))$$

This means that all multicasts will be received in the same view at all operational members of the group. The support of virtual synchrony places several additional obligations on the processes in our system. When the group view changes, all messages delivered at any process in the prior view must be "flushed" out of the system (delivered) before the new view may be used. Further, messages must satisfy a failure atomicity property: if a message m is delivered to any member of a group, and that member stays operational, m must be delivered to all members of the group even if the sender fails before completing the transmission. Note that a message need not be delivered in the view in which it was sent; it suffices that it is delivered in the *same* view at all of its destinations. Note also that not every view change reported by the group membership service need be delivered to the client process.

For demonstrating liveness of our protocols, we will assume that any message sent by a process is eventually received unless the sender or destination fails, and that failures are eventually reported. Recall that these properties are guaranteed by the group membership and message transport services defined in this chapter.

2.6 Conclusion

This chapter has introduced the basic system model to be used in the the rest of this thesis. It has defined the primitive events, explained the underlying services that are available, and specified the algorithms that we wish to implement. In the next chapter, we consider the implementation of fast ordered multicasts in a *single* dynamically changing process group.

Chapter 3

Ordered multicasts in a single process group

3.1 Introduction

This chapter discusses protocols for maintaining causal message delivery order within a single process group[1]. We presents two causal ordering protocols for use in this setting. Both use timestamps to delay messages that arrive out of causal order. The first protocol is non-optimal, in the sense that it sometimes delays messages that do not need to be delayed, and uses extra messages. The second protocol is optimal in message count and delivery delay, but uses larger timestamps. Section 3.3.3 shows how these timestamps can be compressed or even omitted from many of the messages sent in the system, at the cost of some extra overhead messages. Next, we discuss a protocol for totally ordering messages in a single group, and discuss its advantages and disadvantages. Up to this point in the chapter, we do not consider the effect of process failures. Finally, we consider two methods for extending our protocols to provide virtually synchronous message deliveries in

[1]This chapter is based, in part, on work done jointly with Ken Birman and Andre Schiper [BSS90]. However, we substantially extend the results of that paper.

the presence of group membership changes, including membership changes induced by process failure. The subsequent chapters describe several schemes for extending and combining these protocols to work in systems with multiple, overlapping, dynamically changing process groups.

3.2 Logical Timestamps

We begin by describing two methods for assigning timestamps to messages, and for comparing timestamps. They are based on protocols that have previously appeared in the literature. The protocols are standard except in one respect: whereas most timestamping protocols count arbitrary "events", the ones defined here count only *send* events.

3.2.1 Lamport time

The first timestamping protocol is based on one introduced by [Lam78], called the *Lamport clock* protocol. Each process p maintains an unbounded local counter, $LT(p)$, which it initializes to zero. For each event $send(m)$ at p, p sets

$$LT(p) = LT(p) + 1$$

Messages are timestamped with the sender's incremented counter. A process p receiving a message with timestamp $LT(m)$ sets $LT(p) = max(LT(p), LT(m))$. As in [Lam78], one can show that if $send(m) \rightarrow send(m')$ then $LT(m) < LT(m')$. The converse, however, does not hold: the protocol may place a spurious ordering on messages that were sent concurrently. Note that the LT counter for a process is updated at the *rcv* event, as opposed to the *deliver* event, for an incoming message. We make use of this property in the development below.

3.2.2 Vector time

A second timestamping protocol is based on the substitution of *vector times* for the local counters in the LT protocol. Vector times were proposed originally in [Mar84]; other researchers have also used them [Fid88,Mat89,LLS90,Sch88,WPE+83]; our use of them is motivated by a protocol presented in [SES89]. In comparison with Lamport timestamps, these timestamps have the advantage of representing the relation $\rightarrow$ precisely.

A vector time for a process p_i, denoted $VT(p_i)$, is a vector of length n (where n is the number of processes in the group), indexed by process-id. Intuitively, $VT(p_i)[j]$ represents the number of messages sent *by* process p_j whose causal effect has been observed *at* process p_i.

1. When p_i starts execution, $VT(p_i)$ is initialized to zeros.

2. For each event $send(m)$ at p_i, $VT(p_i)[i]$ is incremented by 1.

3. Each message sent by process p_i is timestamped with the incremented value of $VT(p_i)$.

4. When process p_j delivers a message m from p_i containing $VT(m)$, p_j modifies its vector time in the following manner:

$$\forall k \in 1..n: \ VT(p_j)[k] = max(VT(p_j)[k], VT(m)[k])$$

Rules for comparing vector times are:

1. $VT_1 \leq VT_2$ iff $\forall i: \ VT_1[i] \leq VT_2[i]$

2. $VT_1 < VT_2$ if $VT_1 \leq VT_2$ and $\exists i: \ VT_1[i] < VT_2[i]$

Notice that in contrast to the rule for $LT(p)$, $VT(p)$ is updated at the *deliver* event for an incoming message. We will make use of this distinction below.

It can be shown that given messages m and m', $send(m) \rightarrow send(m')$ iff $VT(m) < VT(m')$ [Mat89,Fid88]; that is, vector timestamps represent causality precisely. This is a fundamental property of vector timestamps, and the primary reason for our interest in such timestamps as opposed to Lamport timestamps.

3.3 Causal message delivery

Recall that if processes communicate using CBCAST, all messages must be delivered in an order consistent with causality. Suppose that a set of processes P communicate using only multicasts to the full set of processes in the system; that is, $\forall m : dests(m) = P$. This hypothesis is unrealistic, but Chapter 4 will adapt the resulting protocol to a setting with multiple process groups.[2] We now develop two *delivery protocols* by which each process p receives messages sent to it, delays them if necessary, and then delivers them such that:

$$send(m) \rightarrow send(m') \Rightarrow deliver(m) \rightarrow deliver(m').$$

3.3.1 LT protocol

Our first solution to the problem is based on Lamport clocks; and is referred to as the LT protocol from here on. It is related to other solutions that have appeared in the literature [Lam78,CASD86] and a variant will be used as a building block in Chapter 5. The basic technique will be to delay delivery of a message m with timestamp t until messages with with timestamps greater than or equal to t have been received from every other process in the system, and then deliver messages in timestamp order. Since messages from each other process are received in FIFO order, all messages causally preceding m will be delivered before m. However, since this would only work if every process sends an infinite stream of multicasts, a channel flushing mechanism is introduced to avoid potentially unbounded delays.

[2] This hypothesis is actually used only in the VT delivery protocol.

1. Before sending message m, process p_i increments $LT(p_i)$ and then timestamps m.
2. On receiving message m, process p_j sets $LT(p_j) = max(LT(p_j), LT(m))$. Then, p_j delays m until for all $k \neq i$, the channel between p_j and p_k has been flushed for time $LT(m)$. Note that p_j does not delay messages received from itself, since the channel is trivially "flushed".
3. If m has the minimum timestamp among messages satisfying 2, m may be delivered.

Figure 3.1: The single-group LT CBCAST algorithm

Channel Flushing

Say that the channel from process p_j to p_i has been *flushed at time* LT(m) if p_i will never receive a message m' from p_j with $LT(m') < LT(m)$. Flushing can be achieved by *pinging*. To ping a channel, p_i sends p_j a timestamped inquiry message inq, but without first incrementing $LT(p_i)$. On receiving such an inquiry, p_j, as usual, sets $LT(p_j) = max(LT(p_j), LT(\text{inq}))$ and replies with an ack message containing $LT(p_j)$, without further modifying $LT(p_j)$. On receiving the ack, p_i, as usual, sets $LT(p_i) = max(LT(p_i), LT(\text{ack}))$. If no new messages are being multicast, pinging advances $LT(p_i)$ and $LT(p_j)$ to the same value. Since the channels are FIFO, when p_i sets its clock to $max(LT(p_i), LT(\text{ack})$, it will receive no further messages from p_j with a timestamp less than this value. Thus, the channel between the two processes has been flushed.

The Protocol

The causal multicast protocol is shown in Figure 3.1. Step 1 gives each message from the same process a distinct and monotonically increasing logical timestamp. Step 2 maintains the logical timestamp, and delays a message until no messages with a smaller logical timestamp can be received. The final step delivers messages in causal order. We now proceed to prove this.

Theorem 3.1 *The algorithm in Figure 3.1 always delivers messages in causal order.*

Proof: To prove that causal delivery is achieved, consider two messages m_1 and m_2 such that $send(m_1) \rightarrow send(m_2)$, and hence $LT(m_1) < LT(m_2)$. There are two cases:

1. *The same process sends m_1 and m_2.* Because communication is FIFO, m_1 will be received before m_2, and because $LT(m_1) < LT(m_2)$, step 3 guarantees that m_1 will be delivered before m_2.

2. *Different processes send m_1 and m_2.* According to step 2, m_2 can only be delivered when all channels have been flushed for $LT(m_2)$. As communication is FIFO, and $LT(m_1) < LT(m_2)$, it follows that m_1 has been received. Step 3 then guarantees that m_1 will be delivered before m_2.

□

We now show that the algorithm will deliver every message sent, i.e. that it is live.

Theorem 3.2 *The algorithm of figure 3.1 will eventually deliver every message sent by the client in the absence of process failures.*

Proof: The algorithm delays messages in step 2 or step 3. Since we assume that there are no failures, the channel flushes in step 2 will always terminate. Once step 2 has terminated for a message m, all the messages that step 3 will deliver before m will have already arrived. Therefore step 3 will only delay m until these other messages have been delivered, and m will eventually be delivered. □

The communication of the previous algorithm, however, is high: $2n-3$ messages may be needed to flush channels for every message delivered, hence to multicast one message, $O(n^2)$ messages could be transmitted. For infrequent multicasting, this cost may well be tolerable; the overhead would be unacceptable if incurred frequently. However, in place of pinging, processes can periodically multicast their Lamport timestamps to all other group members. Receipt of such a multicast flushes the channels: at worst, a received message will be delayed until the recipient has multicast its timestamp and all other processes have done a subsequent timestamp multicast. The overhead of the protocol can now be tuned for a given environment, by varying the frequency of flush events.

Readers familiar with the Δ-T real-time protocols of [CASD86] will note the similarity between that protocol and a periodically flushed version of the LT protocol. Clock synchronization (on which the Δ-T scheme is based) is normally done using periodic multicasts [ST87,Cri88,HSSD84]. This modification recalls suggestions made in [Lam78], and makes Lamport clocks behave like weakly synchronized physical clocks. Clock synchronization algorithms with good message complexity are known, hence substitution of a Δ-T based protocol for the Lamport clock-based protocol is an intriguing direction for future study.

3.3.2 VT protocol

A much cheaper solution can be derived using vector timestamps; we will refer to this as the *VT* protocol. The idea is basically the same as in the *LT* protocol, but because $VT(m)[k]$ indicates precisely how many multicasts by process p_k precede m, a recipient of m will know precisely how long m must be delayed prior to delivery; namely, until $VT(m)[k]$ messages from p_k have been delivered. Since $\rightarrow$ is an acyclic order accurately represented by the vector time, the resulting delivery order is causal and deadlock free.

The protocol is shown in Figure 3.2. Step 2 is the key to the protocol. This guarantees that any message m' transmitted causally before m (and hence with $VT(m') < VT(m)$) will be delivered at p_j before m is delivered. An example in which this rule is used to delay delivery of a message appears in Figure 3.3. The

1. Before sending m, process p_i increments $VT(p_i)[i]$ and timestamps m.
2. On reception of message m sent by p_i and timestamped with $VT(m)$, process $p_j \neq p_i$ delays m until

$$\forall k : 1...n \begin{cases} VT(m)[k] = VT(p_j)[k] + 1 & \text{if } k = i \\ VT(m)[k] \leq VT(p_j)[k] & \text{otherwise} \end{cases}$$

   Process p_j need not delay messages received from itself.
3. When a message m is delivered, $VT(p_j)[i]$ is incremented (this is simply the vector time update protocol from Section 3.2.2).

Figure 3.2: The single-group VT CBCAST algorithm

correctness of the protocol will be proved in two stages. We first show that causality is never violated (safety) and then we demonstrate that the protocol never delays a message indefinitely (liveness).

Theorem 3.3 *The* LT *algorithm 3.3.2 always delivers messages in causal order.*

Proof: Consider the actions of a process p_j that receives two messages m_1 and m_2 such that $send(m_1) \rightarrow send(m_2)$.

Case 1. m_1 and m_2 are both transmitted by the same process p_i. Recall that we assumed a lossless, sequenced communication system, hence p_j receives m_1 before m_2. By construction, $VT(m_1) < VT(m_2)$, hence under step 2, m_2 can only be delivered after m_1 has been delivered.

Case 2. m_1 and m_2 are transmitted by two distinct processes p_i and $p_{i'}$. We will show by induction on the messages received by process p_j that m_2 cannot be delivered before m_1. Assume that m_1 has not been delivered and that p_j has received k messages.

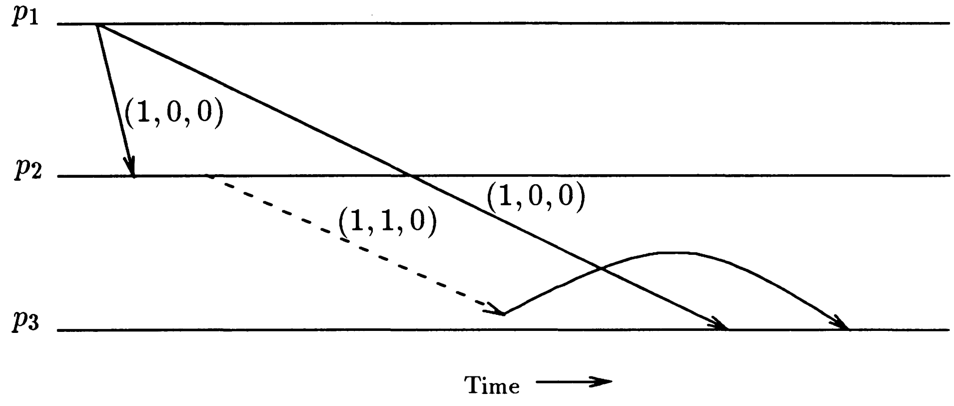

Figure 3.3: Using the VT rule to delay message delivery

Observe first that $send(m_1) \rightarrow send(m_2)$, hence $VT(m_1) < VT(m_2)$ (basic property of vector times). In particular, if we consider the field corresponding to process p_i, the sender of m_1, we have:

$$VT(m_1)[i] \leq VT(m_2)[i] \tag{3.1}$$

Base case. The first message delivered by p_j cannot be m_2. Recall that if no messages have been delivered to p_j, then $VT(p_j)[i] = 0$. However, $VT(m_1)[i] > 0$ (because m_1 is sent by p_i), hence $VT(m_2)[i] > 0$. By application of step 2 of the protocol, m_2 cannot be delivered by p_j.

Inductive step. Suppose p_j has received k messages, none of which is a message m such that $send(m_1) \rightarrow send(m)$. If m_1 has not yet been delivered, then

$$VT(p_j)[i] < VT(m_1)[i] \tag{3.2}$$

This follows because the only way to assign a value to $VT(p_j)[i]$ greater than $VT(m_1)[i]$ is to deliver a message from p_i that was sent subsequent to m_1, and such a message would be causally dependent on m_1. From relations 3.1 and 3.2 it follows that

$$VT(p_j)[i] < VT(m_2)[i]$$

By application of step 2 of the protocol, the $k+1$'st message delivered by p_j cannot be m_2.

□ We now show that the algorithm is live.

Theorem 3.4 *The VT algorithm 3.3.2 will always deliver every message sent in the absence of failures.*

Proof: Suppose that there exists a multicast message m sent by process p_i that can never be delivered to process p_j. Step 2 implies that either:

$VT(m)[i] \neq VT(p_j)[i]+1$, or

$\exists k \neq i: \; VT(m)[k] > VT(p_j)[k]$

and that m was not transmitted by process p_j. We consider these cases in turn.

1. $VT(m)[i] \neq VT(p_j)[i]+1$, that is, m is not the *next message* to be delivered from p_i from p_j. Since all messages are multicast to all processes and channels are lossless and sequenced, it follows that there must be some message m' sent by p_i that p_j received previously, has not yet delivered, and with $VT(m')[i] = VT(p_j)[i]+1$. If m' is also delayed, it must be under the second case.

2. $\exists k \neq i: VT(m)[k] > VT(p_j)[k]$. Let $n = VT(m)[k]$. The n^{th} transmission of process p_k, must be some message $m' \rightarrow m$ that has either not been received at p_j, or was received and is delayed. Under the hypothesis that all messages are sent to all processes, m' was already multicast to p_j. So it must be already in the communication channel. Since the communication system eventually delivers all messages, we may assume that m' has been received by p_j. The same reasoning that was applied to m can now be applied to m'. The number

of messages that must be delivered before m is finite and $>$ is acyclic, hence this leads to a contradiction.

□

3.3.3 Compressing VT Timestamps

If a group contains a large number of processes, the message overhead resulting from the *VT* timestamps may be rather large. We can use two distinct methods for compressing these timestamps. The first method is based on sending timestamp *changes* instead of entire timestamps. In the second method, client message receipt information is propagated around the system in background messages. This background information can be used to further compress *VT* timestamps.

Sending timestamp changes

Instead of sending a complete *VT* timestamp on each message multicast to the group, each process can simply send the *VT* entries that have changed since it last multicast a message to the group. Step 2 of the VT algorithm (Figure 3.2) is modified so that when a message carrying a compressed timestamp is received at a process p_j from a process p_i, it is delayed until all the messages mentioned in the *compressed* timestamp have been delivered. To see that this is correct, we consider the entries that have been omitted from the compressed timestamp carried on m. Two cases arise: such an entry either represents a process from which p_i has never received a multicast, or it was carried on some previous message m' multicast by p_i. In the first case, the causal obligation is trivially satisfied. In the second case, m and m' must differ in at least one *VT* entry, hence m will not be delivered until m' has been delivered. But, if m is delivered after m', the causal obligations of m' with respect to these omitted entries will not be violated. This timestamp compression method is a simplification of methods that have previously appeared in the literature[HHW89].

Message receipt information

Our second method for timestamp compression uses the observation that once a message m has been delivered to all processes within the group there will never be any need for the delivery of a message message m',where $m \rightarrow m'$, to be delayed for the delivery of m. Therefore any timestamp entry on m' whose sole purpose is to delay delivery of m' until after the delivery of m is superfluous. The observation can be further refined: If the initial message m has been *received* at every process, and every process delivers *apparently* unrelated messages in FIFO order, then timestamp entries related to m can be omitted on any causally subsequent message m'. Such a message m' will be received after m everywhere, and will be delivered after m by the FIFO delivery rule. Thus the causal constraints will not be violated. We say that a message that has been received at every process to which it has been sent is *stable.* Message stability information will also be used in Section 3.5, where it plays a central role; this motivates our use of it for timestamp compression.

Of course, we must now consider the question of how to find and disseminate such message stability information. Since the message transport layer will know whether or not a message has been received at all its destinations, the original sender of the message may be able to obtain this information from the transport layer. The information can then be piggybacked on subsequent outgoing messages. A process that is idle for a period of time will multicast the message stability information. Such multicasts do not have to be delivered reliably. A similar technique is used in [HHW89].

Both of our timestamp compression methods can be summed up in the following rules, which modify the VT algorithm (Figure 3.2):

1. If the i^{th} field of VT has not changed since the previous multicast, it can be omitted.

2. If all locally delivered messages from p_i are stable, the i'th field of VT can

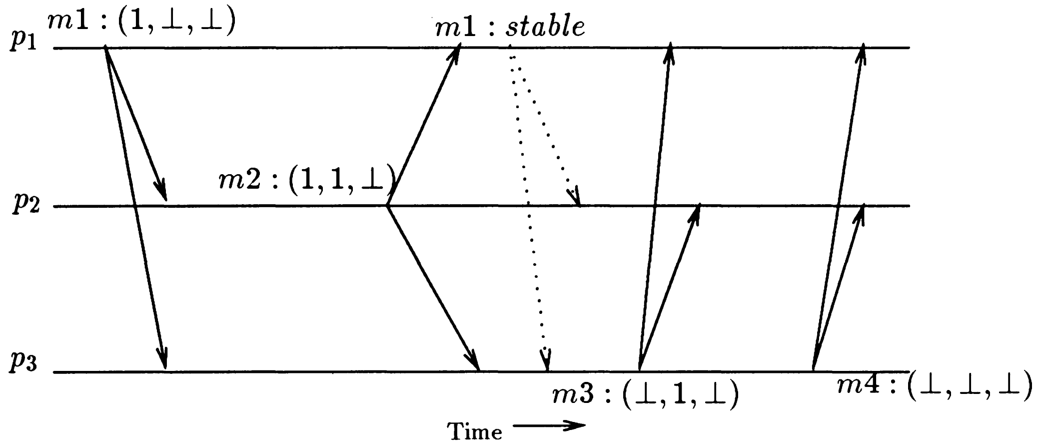

Figure 3.4: Timestamp compression in action

be omitted from outgoing multicasts.

3. Deliver apparently unrelated messages in FIFO order.

Figure 3.4 gives an example of these rules in action. In the figure, the symbol $\perp$ for a timestamp entry indicates that has been omitted due to one of the timestamp compression rules. First, p_1 multicasts a message m_1. After delivering this message locally, p_2 multicasts m_2. Since m_1 is not yet stable and no message has been multicast since field 1 of VT_{p_2} was updated, p_2 puts a timestamp containing an entry for p_1 on m_2. Next, p_1 multicasts a background message (shown by a dotted line) indicating that m_1 has become stable. This message is received by p_3, which then multicasts m_3. On this message the timestamp entry for p_1 is omitted (due to the stability of m_1), but the timestamp entry for m_2 is included (since it is not yet known to be stable and has not yet been sent out by p_3). Finally, p_3 multicasts m_4, which carries *no* timestamp entries - the entry for p_2 can be omitted since it was sent on m_3.

3.4 An ABCAST protocol

One may wonder if our CBCAST protocol can be extended into a fast ABCAST mechanism. ABCAST is a totally ordered communication protocol: all destinations receive all ABCAST messages multicast in a single group in a single, globally fixed, order.

Recall that in Chapter 2, we introduced three variants of totally ordered multicast. One way to define ABCAST is to say that two ABCAST messages sent to the same *logical address* will be totally ordered, but to make no guarantees about ordering for ABCAST messages sent to different addresses. A stronger specification is to say that regardless of the destination processes, if two ABCAST messages overlap at some set of destinations, they are delivered in the same order. An even stronger specification is to impose the condition that the delivery of all ABCAST messages in the system be consistent with some global total order. The tradeoffs between these alternatives are discussed in more detail in [BCG91]. The general protocol given below is capable of implementing any of these specifications, but configuring it to satisfy the stronger specifications will result in a more expensive protocol.

3.4.1 The general ABCAST protocol

Recall that each process always knows which other processes are up and in the group, from the group membership mechanism of Chapter 2. Associated with each view $view_i(g)$ of a process group g will be set of a *ordering* processes, $order(g) \subset view_i(g)$. When we consider failures, we will see that if one of the ordering processes fails, then a new set of ordering processes becomes defined. The ordering processes will, between them, be responsible for assigning a total ordering to every message that is ABCAST to the group. The set of ordering processes may be of any size, from 1 to the number of processes in the group. It is the choice of ordering processes that determines the precise semantics of ABCAST when there is more than one process group.

Intuitively, the system will maintain an LT timestamp over the totally ordered multicasts. Each ordering process will propose a value for the LT timestamp of each ABCAST message; the final value will be the maximum of the proposed values. Since the LT timestamp will only partially order the ABCAST messages, this partial order will be extended to a total order in any consistent way — for example, using some unique attribute of the process that sent the message. This final value will be called the *priority* of the message. ABCAST messages will then be delivered in this total order.

The algorithm for the failure-free case is give in Figure 3.5. (Discussion of failures is deferred to Section 3.5.) In step 1, the ABCAST message m is distributed to all the recipients. In step 3, each ordering process proposes a new timestamp value, which is relayed to the sender along with the current VT timestamp of the ordering process. The purpose of relaying the VT timestamp is to detect other ABCAST messages that are potentially concurrent with m. The final timestamp value and a maximal VT timestamp are then distributed to all the message recipients, via the sender, in step 4. [3] When a process receives the timestamp value for a message, it must wait until all messages that *could* receive a lower timestamp value have been committed. This is done in two stages. First of all, it waits until all such messages have been received — this is the purpose of the wait in step 5. Then it waits until all these messages have actually received their final timestamp value. This is done in step 6, which also ensures that all messages are finally delivered in timestamp order. Delivery of subsequent client CBCAST message from the sender must be

[3]It is also possible to distribute the timestamp value by having the ordering processes CBCAST them directly, and having each process calculate the final timestamp value independently. This may be advantageous when there are small numbers of ordering processes, since latency will be lower. With large numbers of ordering processes, this will result in many multicasts.

delayed until m is delivered in order to maintain the correct causal ordering with respect to m. This also done here.

We now proceed to prove the correctness of the ABCAST algorithm.

Theorem 3.5 (Safety) *The* ABCAST *algorithm (Figure 3.5) delivers all* ABCAST *messages in the same order at every member of the process group. Furthermore,* ABCAST *and* CBCAST *messages are delivered in the correct causal order.*

Proof: First we show that all ABCAST messages are delivered in exactly the same order at each member of the group. Then we show that *all* messages are delivered in an order consistent with causality.

Consider any two ABCAST messages m and m', such that $t_m < t_{m'}$. Since $t_m < t_{m'}$, and the ordering processes propose message timestamps in strictly increasing order, m must have been delivered before m' at one or more of the ordering processes. Since m was delivered before m' at one of the ordering processes, no process will move m' to the delivery queue until m has been received (step 5). So if a process has m' in its delivery queue, either m is committed at that process, or $m \in PC_{m'}$. In the first case, m will be delivered before m' by the first condition in step 6. In the second case, m' will not be delivered until m has entered the delivery queue, by the second condition in step 6. But when m enters the delivery queue, it will do so with a smaller timestamp than m', and will be delivered first. So all ABCAST messages are delivered in exactly the same order at each member of the group.

We now show that messages are delivered in an order consistent with causality. Clearly, the order in which messages are received by the above algorithm is consistent with causality (since all the messages are CBCAST). We therefore focus on the situations where messages may be delivered to the client out of the order in which they are received by the algorithm. Let us suppose that $m \rightarrow m'$, but the algorithm delivers m' before it delivers m. Four cases arise, depending on whether m and m' are CBCAST or ABCAST. We will show that each case gives rise to a contradiction.

- m and m' are ABCAST. If $m \rightarrow m'$, then m must have been delivered at some process before m' was sent. Therefore every ordering process will receive m' after m, and give m' a higher priority value. So m' will be delivered after m, contradicting our assumption.

- m and m' are CBCAST. If delivery of a client CBCAST is delayed by step 6, then all messages depending on it are also delayed. So this case cannot arise.

- m is CBCAST and m' is ABCAST. Since $m \rightarrow m'$, m will be received at each process before m' and will be delivered before m'.

- m is ABCAST and m' is CBCAST. If m' is sent after m is delivered, clearly it cannot be delivered anywhere before m'. Now consider the case where m and m' are sent by the same process p, before m is delivered at p. In this case, delivery of m' will be delayed until after the delivery of m by step 6 of the algorithm.

In order to show that the ABCAST algorithm is live, we will need the following lemma:

Lemma 3.1 *In the* ABCAST *algorithm, when a message m is put in the delivery queue at a process p, every message m' that with a timestamp $t_{m'}$ such that it is possible that $t_{m'} < t_m$ is in one of the following states:*

- *m' has already been delivered to the client.*

- *m is already in the delivery queue.*

- *$m' \in PC_m$.*

Proof: If $t_{m'} < t_m$, then the initial CBCAST for m' must have been received before that for m at one or more of the ordering processes. By the properties of VT timestamps, a process fielding the ordering CBCAST for m will not proceed beyond the delay in step 5 until m' is already in the holding bag. So when m is moved to the delivery queue, m' will be in one of the three states above. □

Theorem 3.6 (Liveness) *The* ABCAST *algorithm will eventually deliver all* ABCAST *and* CBCAST *messages.*

Proof: Clearly, each ABCAST message m will eventually become committed and eventually enter the delivery queue with some timestamp t_m and some set of potentially concurrent messages PC_m. Since all messages in PC_m will eventually enter the delivery queue, eventually m will no longer be delayed by the first condition in step 6. By Lemma 3.1, the number of messages that must be delivered before m is bounded by the sum of the size of PC_m and the length of the delivery queue at the time that m is placed there. Since both of these are finite, step 6 of the algorithm cannot delay the delivery of m indefinitely. □

There are some interesting points to note about this algorithm. There is no requirement that proposed or committed timestamps are dense — i.e the values need not be successive. We will make use of this fact in developing a multiple-group ABCAST algorithm in Section 4.7. Also, iff the set of ordering processes is the entire set of processes in the current view of the group, this algorithm will flush all CBCAST messages out of the system, such that at the point the ABCAST is delivered, every process will have received exactly the same CBCAST messages. This is one method for implementing the group flush operation, discussed in Section 3.5, where other implementations are also given. Furthermore, this version of the algorithm can be optimized to produce Skeen's ABCAST algorithm as described in [BJ87b]. On the other hand, if there is only one ordering process, this is essentially a token-based ordering algorithm. The single ordering process can multicast the final timestamp directly, which can save some message latency. Such an algorithm is essentially identical to the replicated data protocol proved correct in [BJ89,Sch88].

The cost of sending an ABCAST depends on the locations where multicasts originate, the number and location of the ordering processes, and the circumstances under which the set of ordering processes is changed. If multicasts tend to originate at the same process repeatedly, then it is clearly advantageous to make that site the single ordering process. Then the cost is one CBCAST per ABCAST. This represents a major improvement over the existing ISIS ABCAST protocol, which implements the strongest specification of ABCAST. However, since an ABCAST with a single ordering process satisfies a weaker specification, it can only be used in certain situations. If stronger forms of ordering are required, more than one ordering process may be necessary, as discussed in Section 4.7.

3.5 Flush and GBCAST

Up to now, we have presented our multicast algorithms under the assumption that failures do not occur in the system. We now consider group membership changes, including joins and failures. (The failure case of our solutions can easily be optimized for the case where a process leaves a group voluntarily.) Recall that we assume that the effects of failures are somewhat limited – processes fail only by crashing, and process failures are reported in the same order at each surviving process that is a member of the group by the group membership service. In the absence of failures, messages are always reliably transmitted between any pair of processes.

As with messages, we will distinguish between the *reception* of a new view from the group membership service, and the *delivery* of that view to the client. When the group membership changes, in order for the execution to remain *virtually synchronous*, all the remaining processes in the group must agree on the list of messages to be delivered to each surviving process in the old view *before* the new view is delivered. In general, in order to maintain virtual synchrony, if a message has been delivered at *any* of the surviving processes before the new view is delivered to that process, it must be delivered at *all* of them. Once a new view v has been delivered at a process, subsequent messages from that process should be multicast to all members of the new view, where they should be delivered subsequent to the delivery of v. In addition, the correct causal and total orderings between messages should be maintained, both before and after the delivery of the new view v. If the new view was induced by the failure of a group member p, each surviving process p' will see an execution where it appears that p received all messages that p' did. While this may not in fact be the case, the actual execution is indistinguishable from an execution where the failing process received the messages and immediately failed.

We consider algorithms for maintaining the illusion of virtual synchrony despite group membership changes. The first algorithm is relatively simple, but has the disadvantage of requiring $O(n^2)$ messages each time the group view changes. The second algorithm is more complex, and inhibits the transmission of client multicasts for longer, but it only requires $O(n)$ messages. The first algorithm may be sufficient when process groups are small, or group membership changes infrequently. The second algorithm will be needed in other circumstances.

3.5.1 Message Atomicity

Both algorithms use the same method for ensuring message atomicity – i.e. ensuring that a message received and delivered by any process in a process group is received and delivered by every process in the group. Recall that in Section 3.3.3 we introduced the notion of message *stability*: a message is stable if it has been received at all its destinations. Message stability information for each message m is propagated by the sender of m to every other process in the group. We now modify the message delivery algorithm to maintain a local copy of each message m at each process p until p knows that m is stable. The stability information is propagated by using background messages, as detailed in Section 3.3.3. Clearly, since a stable message has been received at each process, we do not need to be concerned about the atomicity of that message. Hence, the membership change algorithms introduced below work with the *unstable* messages at each site. Every process retains a copy of each message m that it has delivered to the client, until it knows that m is stable. This modification to the basic message delivery algorithm will be assumed below.

3.5.2 Basic Flush Algorithm

To achieve virtually synchronous addressing when group membership changes while multicasts are active, we introduce the notion of *flushing* the communication in a process group. Consider a process group g in group view $view_i(g)$. Say that a new view $view_{i+1}(g)$ is received. There are two cases: $view_{i+1}(g)$ could reflect the addition of a new process, or it could reflect the departure (or failure) of a member. Assume initially that view changes are always due to adding new processes (we handle failures in Section 3.6). We will flush communication by having all the processes in $view_{i+1}(g)$ send a message `"flush i+1 of g"`, to all other members. During the period after sending such messages and before receiving such a flush message from all members of $view_{i+1}(g)$, a process will accept and deliver messages but will not initiate new multicasts.

Because communication is FIFO, if process p has received a flush message from all members of g under view $i+1$, it will first have received any messages sent in view i. It follows that all communication sent prior to and during the flush event was done using VT timestamps corresponding to $view_i(g)$, and that all communication subsequent to installing the new view is sent using VT timestamps for $view_{i+1}(g)$. This establishes that multicasts will be virtually synchronous in the sense of Section 2.

3.6 Failure atomicity

What about the case where some member of g *fails* during an execution? Then $view_{i+1}(g)$ will now reflect the departure of some process. This section explains how our flushing algorithm works in the context of CBCAST; the next section extends the algorithm to work with ABCAST.

Assume that process p_j has received a message m that was multicast by process p_i. If p_i now fails before completing its multicast, there may be some third process p_k that has not yet received a copy of m. To solve this problem, each process p_j must retain a copy of all messages that it has delivered locally until they are *stable.* If process p_i fails, each other process p_j will multicast the messages it has received from p_i that are not yet stable. Processes identify and reject duplicates.

On receiving $view_k(g)$ indicating that p_i failed, p_j runs this protocol:

1. Close the channel to p_i.

2. For any unstable multicast m initiated by p_i, send a copy of m to all processes in $view_k(g)$ (duplicates are discarded on reception).

3. Send a `flush` message to all processes in $view_k(g)$.

4. Wait until `flush` messages have been received from all processes in $view_k(g)$. If p_l, one of the processes in $view_k(g)$ fails, this incarnation of the flush protocol is suspended until the subsequent incarnation (started by the view change induced by the failure of p_l) is about to terminate. At this point, if a `flush` message has not been received from p_k, it can be safely simulated, and this incarnation of the protocol may continue.

5. Simulate the receipt of any `acknowledgement` messages required by the LT protocol.

6. Discard any messages delayed pending a message from p_i. Deliver all remaining messages in the correct order.

7. Finally, p_j ceases to maintain $VT_g[i]$. Since all messages from the previous view have been received, the remaining VT fields are reset to 0. The new view is delivered to the client as soon as all preceding views have been delivered.

Step 2 ensures atomicity and step 4 prevents deadlock if one of the processes executing the flush protocol fails. Step 5 ensures that the *LT* protocol will complete. Step 6 relates to chains of messages $m_1 \rightarrow m_2$ where a copy of m_2 has been received but m_1 was lost in a failure; this can only happen if every process that received m_1 has failed (otherwise a copy of m_1 would have been received prior to receipt of the `flush` message). In such a situation, m_2 will never have been deliverable and hence can be discarded.

Although this algorithm requires $O(n^2)$ messages, it can be easily transformed into a coordinator-based algorithm requiring $O(n)$ messages. In such an algorithm, the unstable messages and the `flush` messages are sent just to a designated co-ordinator process, which distributes them all to each other process in one atomic message. The failure simulation step has to be extended to take care of the case where the ordinator fails during the execution of the protocol, as follows:

- If the co-ordinator fails *before* distributing the `flush` message, the unstable messages are resent to the co-ordinator in a subsequent incarnation of the flush protocol and this incarnation is discarded.

- If the coordinator fails *after* distributing the `flush` message, the `flush` message will be atomically distributed by a subsequent incarnation of the flush protocol and this incarnation can be normally terminated.

3.6.1 ABCAST Flush Algorithm

The algorithm of the previous section will ensure virtually synchronous causally ordered multicasts. In order to ensure that all ABCAST messages are virtually synchronous, we simply extend the delivery (step 5) step of the basic flush algorithm as follows:

- If an ordering message has not been received for any pending ABCAST, assign it an arbitrary order consistent with causality, using a well-known rule.

- Deliver all pending ABCAST and CBCAST messages in an order consistent with causality.

Since each ordering message sent by the ordering processes will be causally dependent on the previous one, all the surviving processes will have received exactly the same sequence of ordering messages when the flush completes. Also, since they are using a well-known rule to assign order to ABCAST messages for which there is no ordering message, all ABCAST messages will receive exactly the same priority at each process. Thus, they will all be delivered in the same order at each process in the system, and this order will be consistent with causality.

3.7 Conclusion

This chapter has developed two algorithms for maintaining causal delivery ordering in a single process group. The first algorithm has small message overhead but is non-optimal in that it may unnecessarily delay messages. The second algorithm has greater message overhead, but only delays message delivery when absolutely necessary. The message overhead is related to the size of the group; it can be reduced by the addition of extra background messages. If so reduced, the message overhead will be related to the number of recently active processes in the group. Next, we discussed a general algorithm for extending this causal ordering on the multicasts to a total order that also respects causality.

These algorithms go significantly beyond prior work on maintaining virtual synchrony within a single process group; unlike the protocol of [BJ87b], they do not require a garbage collection phase; and unlike the protocols of [LLS90], they do not depend on delayed background messages for correctness, only for performance. Our total ordering protocol subsumes several other protocols that have appeared in the literature, and is easily extensible to the multiple-group case. In the next chapters, we will discuss several methods for extending these single-group algorithms to work in settings with multiple process groups.

1. The sender CBCAST's a proposed ABCAST m to the group.
2. Every process puts m in a *holding bag*, and marks m as *uncommitted.*
3. Each *ordering* process sends a proposed LT timestamp value to the sender of the message, along with its current VT timestamp.
4. The sender computes the final LT timestamp value for m, denoted t_m, by taking the maximum of all the values that the ordering processes proposed for m and appending its process id. It then computes VT_{max}, the maximum over the VT timestamps that each ordering process sent to it. Finally it sends a CBCAST message containing t_m and VT_{max} — the *ordering CBCAST* for m.
5. Each process p, on receiving the ordering CBCAST for m, waits until it has received and processed all messages indicated by VT_{max}. Then p updates the LT timestamp value for m, marks m as *committed*, and moves m to a *delivery queue.* The delivery queue is kept sorted by timestamp. We denote the set of messages in the holding bag at this point as PC_m, which is the set of messages potentially concurrent with m.
6. A message m is delivered to the client when it has the lowest timestamp of all messages in the delivery queue, and all messages in PC_m have been committed and moved to the delivery queue. Any CBCAST that is causally after any message in the delivery queue is delayed until that message has been delivered. No other CBCASTs are delayed.

Figure 3.5: The general ABCAST protocol

Chapter 4

Ordered Multicasts in Overlapping Process Groups

4.1 Introduction

This chapter discusses the problems introduced by ordered multicasts in multiple overlapping process groups. First we discuss some of the problems and issues arising when attempting to maintain causality across such groups. Then we present an underlying framework for developing algorithms that guarantee causal message delivery in this environment. Several different algorithms are then constructed from this framework. These algorithms display different tradeoffs between the overhead on each message, the degree of fault-tolerance provided, the extra delay imposed on each message, and the number of extra messages in the system.

Our underlying framework for multiple-group algorithms builds on algorithms for maintaining causality in *individual* process groups. The choice of algorithms used for this purpose is immaterial, as long as they can be extended to fit into our framework. Consequently, we are able to use our framework to extend several single-group causality algorithms that others have proposed into multiple-group causality algorithms, and we also use it to develop algorithms for both point-to-point messages within groups and client-server groups. We conclude the chapter with a discussion on maintaining totally ordered message deliveries across multiple process groups, i.e. ensuring that the total orders chosen in individual process groups do not conflict with each other in any way. The algorithm we develop to ensure this is a relatively straightforward extension of the single-group total order algorithm developed in the previous chapter.

4.2 Multiple-group Causal Order

The basic problem when maintaining causality across multiple groups is that a message sent in one group may be causally dependent on a message sent in *another* group. Such a causal dependency must be honoured at any process that will receive both messages, i.e. any process that is a member of both groups. An example is given in Figure 4.1. In this figure, message m', sent in group g', is causally dependent on message m, sent in group g. So process p' should deliver m before it delivers message m'. However, a single-group causality algorithm cannot infer a causal connection between m and m', since they were sent in different groups. Note also that the chain of messages forming the causal connection between m and m' could pass through groups other than g and g' and could in fact be *arbitrarily* long. This is one of the main reasons that the problem is difficult to solve.

The example above also raises a tricky issue of fault-tolerance; if p sends a message m to group g, sends a message outside the group, and then fails, the single-group recovery algorithm is under no obligation to deliver m to g (since it might not have been received at any process save p in g). We will discuss this issue further in Section 4.4. However, since the single-group algorithms will hide process failures inside a single group, the only other observable effect of failures at the multiple-group level is the total disappearance of a group. This possibility does not affect our algorithms or proofs, hence we need not mention failures further other than in Section 4.4.

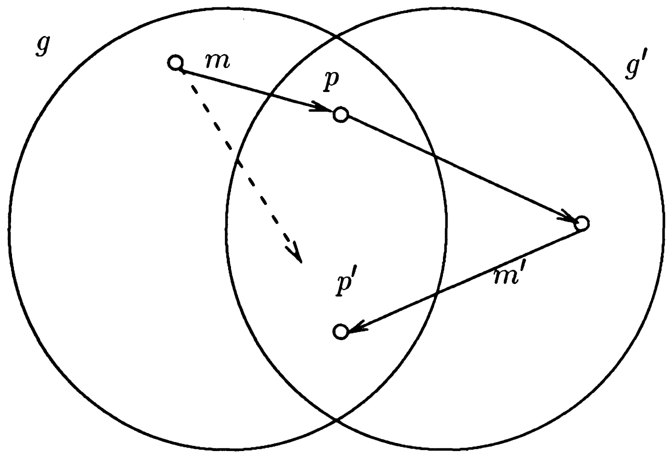

Figure 4.1: Causal ordering across multiple process groups

At first sight, the multiple-group causality problem resembles the single-group causality problem. In fact, one might think that the multiple-group problem can be solved by an appropriate generalization of the single-group problem, by simply identifying processes in the single-group setting with groups in the multiple-group setting. However, it differs in several important ways:

- Within a single process group, we assumed the existence of a group membership service; in particular, each process in the group knew all other processes with which it would be communicating. Since we wish to be able to scale our protocols to run in systems with very large numbers of groups, we do not wish to make the same assumption in the multiple-group setting; there is no global list of groups, and the creation and deletion of a group is known only to the processes that are members of that group.

- Within a single process group, all messages are sent to every member of the group. In the multiple-group setting, each message is sent only to one group.

- The overall number of process groups is likely to be large. In a single process group, either the number of processes is small or, if it is large, the message traffic is likely to follow certain patterns that we can take advantage of.

Some of these differences make the multiple-group problem harder than the single-group problem; others make it easier. Overall, they are incomparable. The next section develops a general framework for algorithms that can recognize and honour causal obligations across multiple groups.

4.3 A general multiple-group CBCAST algorithm

In this section, we develop a general algorithm for maintaining causality in multiple overlapping groups. The algorithm is general in that it does not depend on a particular algorithm being used to maintain causality *inside* any particular process group. The basic idea is as follows: whenever a message chain passes from a process group g to another process group g' (via a process p that is a member of both g and g'), "group g" should piggyback sufficient information on the outgoing message to ensure that if the chain ever re-enters g in the future, all causal obligations can be satisfied. This means that all messages sent in group g that have been delivered to the client at p before it multicasts m to g' should be delivered to any process $p' \in g$ before any message causally subsequent to m is delivered at p'. The information that is piggybacked to ensure this ordering is called the *external timestamp* for g. Whenever a process p' that is a member of group g receives a message m containing t_g, an external timestamp for g, from some other group g', it must ensure that all messages that the external timestamp "represents" have been delivered before it delivers m. This is made more precise below.

Clearly, each group in the system may have to maintain and transmit external timestamps for other groups. Our multiple-group algorithm will work with any single-group causality algorithm for which such external timestamps can be defined. We now give a formal definition of the required properties of external timestamps, and the required extensions to single-group causality algorithms.

4.3.1 Properties of external timestamps

We must be able to extend our single-group algorithms with three new primitives, *tstamp*, *sync* and *delivered*. The *tstamp* primitive returns an external timestamp for a process group. The i^{th} request for an external timestamp by process p in group g is modeled by the primitive event $ts.req_p^i(g)$; the completion of this operation, returning timestamp t is modeled by the primitive event $ts.return_p^i(t,g)$. Similarly, if t is an external timestamp for group g, then a process p can synchronize with the timestamp by invoking the $sync_p(t,g)$ primitive. The i^{th} invocation is modeled by the primitive $sync.req_p^i(t,g)$. This operation will not return until all messages in the causal past of the timestamp request that returned t have been delivered to the client. This completion operation is modeled by primitive event $sync.return_p^i(g)$. Since the single-group algorithm does not have access to message delivery information (at this level of abstraction it is just passing messages to the multiple-group algorithm), the multiple-group algorithm should call $delivered(m,g)$ when it finally delivers to the client a message m that was sent in group g. The required relationship between a *tstamp* primitive and the corresponding *sync* primitive can be formally stated as follows:

$$\begin{aligned}
&\forall g, \forall p, p' \in g: \\
&\quad delivered_p(g,m) \xrightarrow{\mathrm{p}} ts.req_p^i(g) \xrightarrow{\mathrm{p}} ts.return_p^i(t,g) \rightarrow sync.req_{p'}^j(t,g) \Rightarrow \\
&\quad delivered_{p'}(g,m) \xrightarrow{\mathrm{p'}} sync.return_{p'}^j(g) \qquad (4.1)
\end{aligned}$$

We require that two external timestamps can be tested for equality. In addition, in order to minimize the number of external timestamps that are maintained in the system, we require that a *merge* operation be defined on pairs of timestamps for the same group. If a message is in the causal past of *either* of two external timestamps t and t', then it is in the causal past of $merge(t, t')$. Formally:

$$\begin{aligned}
&\forall g, \forall p, p', p'' \in g : \\
&\quad delivered_p(g,m) \xrightarrow{\mathrm{P}} ts.req_p^i(g) \xrightarrow{\mathrm{P}} ts.return_p^i(t,g) \rightarrow sync.req_{p''}^k(merge(t,t'),g) \bigvee \\
&\quad delivered_{p'}(g,m) \xrightarrow{\mathrm{P}'} ts.req_{p'}^j(g) \xrightarrow{\mathrm{P}'} ts.return_{p'}^j(t',g) \rightarrow sync.req_{p''}^k(merge(t,t'),g) \Rightarrow \\
&\quad delivered_{p''}(g,m) \xrightarrow{\mathrm{P}'} sync.return_{p''}^k(g) \qquad (4.2)
\end{aligned}$$

We can immediately define a partial order on external timestamps:

$$t' \leq t \Leftrightarrow merge(t', t) = t$$

We now formally define what it means for a message m to precede an external timestamp t:

$$\exists t', p \in g : (t' \leq t) \bigwedge delivered_p(g,m) \xrightarrow{\mathrm{P}} ts.req_p^i(g) \xrightarrow{\mathrm{P}} ts.return_p^i(t', g)$$

This implies that if m precedes t and a *sync* operation is invoked with some timestamp $t' \geq t$, then m will be delivered before the *sync* operation completes.

Finally, we require that the *tstamp* and *sync* operations always complete, i.e. that they are live. Completion of the *tstamp* operation is unconditional; completion of the *sync* operation clearly depends on the appropriate messages being delivered. Formally:

$$ts.req_p^i(g) \xrightarrow{\mathrm{P}} ts.return_p^i(t,g) \qquad (4.3)$$

$$\begin{aligned}
&ts.return_p^i(t,g) \rightarrow sync.req_{p'}^j(t,g) \bigwedge \\
&\quad (\forall m : delivered_p(g,m) \xrightarrow{\mathrm{P}} ts.return_p^i(t,g) : \\
&\qquad delivered_{p'}(g,m) \xrightarrow{\mathrm{P}'} sync.return_{p'}^j(g)) \qquad (4.4)
\end{aligned}$$

Intuitively, a *tstamp* operation at a process p in a process group g returns a value identifying a consistent cut [CL85] in the process group; if this value is passed to a *synchronize* operation at any other member of the process group, the *synchronize* operation will not complete until that member had reached the cut represented by t. These primitives can be constructed for many common single-group algorithms in the literature; we give details in Section 4.4. However, there are several design choices in the construction of these primitives and these choices can have a significant effect on the characteristics of the final overall algorithm. We will discuss this further in Section 4.4.

4.3.2 The canonical multiple-group algorithm

Our multiple-group algorithm works as follows. Each process runs a single-group algorithm for each of the process groups of which it is a member. Each process group can run a different algorithm as long as all processes that are members of the group know the algorithm that is being used to maintain causality in that group, and the algorithm is extended to provide the operations defined in Section 4.3.1. Client *send* requests are processed first by the multiple-group algorithm, and then passed on to the the single-group algorithm for the appropriate group, obtaining external timestamps where necessary. Similarly, when a message is delivered by one of the single-group algorithms, it is received by the multiple-group algorithm, all external timestamps are honoured, and the message is finally delivered to the client only when it is safe to do so. The resulting layering of our protocols is illustrated in Figure 4.2. The solid lines represent the paths taken by client messages, and the dotted lines represent requests to obtain or honour external timestamps.

The general multiple-group algorithm is given in Figure 4.3. It maintains a list of external timestamps, initialized to the empty list. We call this the *external timestamp list.* The external timestamp list may contain up to one external timestamp for each group in the system (we will discuss methods for reducing the size of the list in Chapter 5). Each external timestamp in the list has associated with it a list of local groups to which it has been sent — the *forwarded* list. When the client

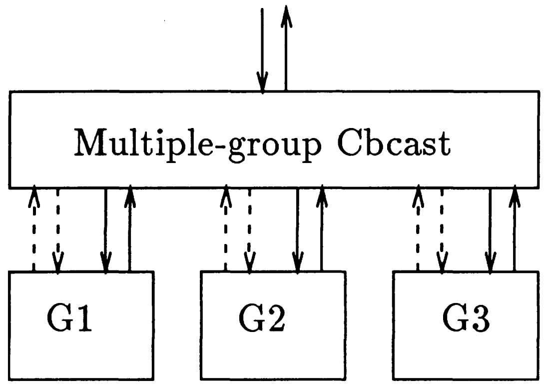

Figure 4.2: Layering in the multiple-group algorithm

wishes to send a message m to a group g of which it is a member, the algorithm gets a new external timestamp from each group that it has interacted with *since* it last sent a message to g. These timestamps are appended to m when it is multicast to g. When a message carrying external timestamps is received, the algorithm ensures that all causal obligations relating to this message have been honored by performing *sync* operations for the new external timestamps where necessary.

We now proceed to the proof of correctness of the algorithm. Consider any pair of causally related messages in the system, m_1 and m_l, where $m_1 \rightarrow m_l$. In order to show that the above algorithm is correct, we must show that m_l is delivered everywhere causally after m_1. If the destinations of m_1 and m_l do not overlap, causality is trivially satisfied. It remains to show that m_1 is delivered before m_l at all overlapping destinations. In general, there will be a chain of messages forming the causal connection between m_1 and m_l; we will denote such chains as follows:

$$p_1 \overset{m_1}{\underset{g_1}{\rightsquigarrow}} p_2 \overset{m_2}{\underset{g_2}{\rightsquigarrow}} p_3 \overset{m_3}{\underset{g_3}{\rightsquigarrow}} \ldots . p_{l-1} \overset{m_{l-1}}{\underset{g_{l-1}}{\rightsquigarrow}} p_l \overset{m_l}{\underset{g_l}{\rightsquigarrow}} p_{l+1} \tag{4.5}$$

This schema signifies that process p_1 multicasts message m_1 to group g_1, that process p_2 first receives message m_1 as a member of group g_1 and subsequently multicasts m_2 to g_2, and so forth. In general, g_i may be the same as g_j for $i \neq j$ and p_i and p_j may be the same even for $i \neq j$ (in other words, the processes p_i and the groups g_i are not necessarily all different). However, without loss of generality, in the rest of the chapter, we will assume that all the processes $p_1 ... p_{l+1}$ and groups $g_1 ... g_l$ are distinct. Since the destinations of m_1 and m_l overlap, p_{l+1} must be a member of g_1. We denote by $t_p(m, g)$ an external timestamp value returned at some process $p \in g$ after m has been delivered at that process. We omit the subscript where the context is clear.

Lemma 4.1 *In any message chain $m_1 ... m_l$, where $m_1 \rightarrow m_l$, m_1 is sent by process p_1 in group g_1, m_l is sent by process p_l in group g_l and $l \geq 2$, where l is the length of the message chain, p_{l+1} will receive from p_l a message carrying an external timestamp $t(g_1)$ for g_1 before it delivers m_l to the client, where $t(g_1) \geq t(m_1, g_1)$.*

Proof: We prove the lemma by induction on l, the length of the message chain. In the base case, $l = 2$. By step 1 of the send algorithm, process p_2 will perform a *tstamp* operation on g_1 sometime after it received m_1 and before it sends m_2. This *tstamp* operation will return a value $t(g_1) \geq t_{p_2}(m_1, g)$, and the external timestamp list at p_2 will contain some timestamp t' for g_1 where $t' \geq t(g_1) \geq t_{p_2}(m_1, g_1)$. If this value has not been sent to g_2 before m_2 is sent, it will be sent piggybacked on m_2, by steps 2 and 4 of the send algorithm. This proves the base case. Turning to the induction case, we assume the lemma is true for all message chains of length l' less than l. In particular, in a message chain of length l, when p_l receives m_{l-1}, it will have received an external timestamp for group g_1 with the required properties. By applying a similar argument to the base case, we can conclude that this timestamp will have been forwarded to p_{l+1} by the time that m_l is received there. i.e. before m_l is delivered to the client. □

Theorem 4.1 *The basic multiple-group algorithm delivers messages in causal order.*

Proof: Consider any chain of messages $m_1 \rightarrow m_l$ where m_1 and m_l have overlapping destinations and all the groups on the chain are distinct. Since m_1 and m_l have overlapping destinations, p_{l+1} must be a member of g_1. By Lemma 4.1, p_{l+1} will receive from p_l an external timestamp t for g_1, where $t \geq t(m_1)$, before it delivers m_1 to the client. Two cases now arise.

- If the first such timestamp was received piggybacked on m_l, then by step 1, p_{l+1} will complete $sync(t, g)$ before delivering m_l. By the properties of *sync*, all messages that causally preceded m_l already will have been delivered to the client before the *sync* operation completes.

- If the first such timestamp was piggybacked on some previous message m', from p_l, then m' will not be delivered until all messages causally previous to m_l have been delivered. But m_{l+1} will be delivered after m', ensuring causal order.

□

We now consider liveness; i.e. we show that all messages are eventually delivered.

Theorem 4.2 *(Liveness) The basic multiple-group algorithm eventually delivers all messages sent by clients.*

Proof: Suppose the contrary: there is a message that a client sends that can never be delivered. Since *tstamp* operations always terminate, every message that the client sends will always be passed to the single group algorithm for the appropriate process group. Since we assume that the single-group algorithms are live, every message will eventually be delivered by the single-group algorithm. Therefore, the only place messages can be indefinitely delayed is in the receive algorithm, if one of the required *sychronize* operations does not terminate. Since these operations are guaranteed to terminate provided that *deliver* operations are invoked for all causally preceding messages, the only way a message can never be delivered is if there exists some causally preceding message that can also never be delivered. Clearly, this would require an infinite descending chain of undeliverable messages. However, each message can only be causally preceded by a finite number of other messages, hence we have a contradiction. □

4.4 Implementing external timestamps

We now discuss several methods for implementing external timestamps with the properties required by the multiple-group causal delivery algorithm. This section discusses two implementations of external timestamps for the single-group causal multicast algorithm of Chapter 3. Our first implementation is relatively expensive and completely fault-tolerant. We then discuss a much cheaper implementation of external timestamps, which incurs some fault-tolerance risk. We believe that the level of risk incurred is acceptable in many environments. Finally, we briefly discuss implementations of external timestamps for two other causal multicast protocols that have appeared in the literature — the protocols of Peterson [PBS89] and Ladin [LLS90].

4.4.1 Conservative timestamping scheme

Recall that an external timestamp t for a process group g is supposed to represent enough of the causal history of that process group so that when t is presented at any process in the group, all causal obligations can be satisfied, i.e. all messages prior to t can be delivered before the *sync* operation terminates. One way to ensure this, of course, is to ensure that all the relevant causal obligations have been honoured *before* the *tstamp* operation terminates. That is, a *tstamp* operation at process p will terminate when all messages that were delivered at p before the *tstamp* operation was invoked have been received at all their destinations. Since the *VT* algorithm of Chapter 3 maintains this information for timestamp compression and fault-tolerance purposes, it is easy to implement the *tstamp* operation — it simply waits until all causally preceding messages are *stable.* All *tstamp* operations on the group simply return the same, null, value. The *sync* operation waits until all non-causally related messages previously received at this client have been delivered. This is because some of these messages may in fact be causal predecessors of the message carrying the timestamp. The *tstamp* operation will guarantee that such causal predecessors have already been received. Clearly, these implementations of the basic operations satisfy the correctness conditions 4.1, 4.2 and 4.4, despite the presence of failures in the system. Therefore the overall algorithm is correct.

Since all external timestamps for a given group have the same value, the basic multiple-group algorithm can be optimized to remove all timestamp comparison and update rules for process groups that use the conservative external timestamping scheme. Therefore, messages need not piggyback external timestamps for such process groups. This means that *sync* must always be performed whenever a message chain enters a group; however, since it is a null operation, this is not a problem. It is easy to see that these optimizations do not affect the correctness of the algorithm. If all process groups in the system use the conservative timestamping scheme, we say that the system is running the *Conservative Algorithm.*

4.4.2 Optimistic external timestamping scheme

The conservative scheme of the previous section has one major drawback — in some environments, the delay for a *tstamp* operation could be quite large. In the "optimistic" scheme, we take a different approach to ensuring that each process group is able to meet the causal obligations induced by its external timestamps. We simply use the internal group representation of logical time. A *tstamp* operation for process p in group g will return immediately, with a timestamp consisting of:

- v, The view number of group g.
- vt, A vector timestamp for group g.

Two such timestamps t_1 and t_2 are merged as follows:

1. If one of the view numbers is greater than the other, the result is the timestamp with the greater view number.
2. Otherwise, the result is a timestamp with $v = v_{t1} = v_t2$, $vt = merge(vt_{t1}, vt_{t2})$.

The *sync* operation for timestamp t consists of:

1. If the view number $t.v$ in the timestamp is less than the current view number of the group, return immediately.
2. Otherwise, wait until $t.vt$, the vector time in the timestamp is less than the current vector time for the group.

There is one problem with this approach. Since the *tstamp* operation returns without waiting for the messages contained by the timestamp t to become stable, it is possible for all copies of a message m to disappear from the system even though m is contained by a live timestamp t. This can arise if a process p sends a message m within a group g, sends another message m' within a group g', and then fails. It is possible that m never reaches any correct process in g, even though it is contained by t_g, the external timestamps for g that was piggybacked on m'. Note that it would not be sufficient to extend the external timestamp to contain the text of the message m, since the failure/view change algorithm will only look for undelivered messages in processes that are members of g. Conceptually, there are two approaches to solving this problem:

- Put the text of unstable messages into external timestamps, and extend the view change algorithm to look for undelivered messages *outside* the group. This would result in a protocol very close to that presented in [BJ87b].

- Ensure that all messages become stable with arbitrarily high probability.

While the first alternative is cleaner, it will result in view change protocols that must potentially run at every process in the system each time any group changes membership — since there may be an external timestamp for that group, containing a message that that group needs, at any process in the system. So such a system will not scale well. In addition, since external timestamps are large, it becomes imperative to aggressively garbage-collect them, as in [BJ87b]. While this garbage collection is not overly expensive in many situations, it does add extra complexity to the protocol.

There are several methods of achieving the second alternative, making sure that messages become stable with high probability. A non-exhaustive list includes:

- Build a k-resilient protocol that operates by delaying communication outside a group until all causally previous messages have been delivered to k sites; for $k = n$, this reverts to the conservative approach.

- Rely on the underlying message transport protocol to deliver the message with high probability to enough destinations to survive any pattern of failures.[1]

- Construct a special-case message logging server which is always guaranteed to have a copy of any messages that are not yet stable, as in [Pow83].

The solution selected will depend on the system designer's tolerance for potential causality violations. It should be noted that limitations such as this are common

[1]For example, a transport facility that exploits lower-level multicast capabilities. With the development of programmable communication boards, this is an increasingly realistic alternative.

in distributed systems; a review of such problems is included in [BJ89]. In fact, most industrial projects are considered "fault-tolerant" if $k = 1$!

Now, if we assume that all messages referenced in external timestamps eventually become stable, it is easy to show that the implementation of external timestamps given at the beginning of the section satisfies conditions 4.1, 4.2 and 4.4, despite the presence of failures in the system. Therefore, a multiple-group algorithm based on this implementation of external timestamps is correct. Unlike the conservative algorithm, no delays are incurred when performing a *tstamp* operation; however timestamps are large and cannot, in general, be optimized away.

4.4.3 External Timestamps for other causal protocols

The protocols of Ladin [LLS90] and Peterson [PBS89] also provide causal ordering within limited domains. Ladin's protocol will causally order client/server messages for a single replicated service; it uses a vector timestamp which contains one entry for each server that runs a copy of the service. In order to extend the protocol to provide causal ordering in a multiple client/server setting, external timestamps can be implemented simply using the service vector timestamp. The multiple-group algorithm should be run at each server, and clients should be extended to pass external timestamps from one group to another. The details of the algorithm are similar to our optimistic algorithm above.

Peterson's Psync protocol provides causal ordering within a "conversation", which corresponds to our notion of a process group [2]. Messages are always delivered in causal order. Each process explicitly maintains a view of the "context graph" of the conversation, where the context graph encodes the message history of the conversation in a partial ordering. Stability information can easily be inferred from this view. Using our techniques, we can extend Psync to provide causal ordering

[2] For present purposes. Conversations do not of themselves provide virtual synchrony.

across multiple conversations. An external timestamp for a Psync conversation is simply a list of unstable messages for that conversation. The $sync_p(t, C)$ operation for a process p in a conversation C is straightforward; simply insert all messages in t that are not already in the local view of C into it. This extension would be straightforward to implement.

Note that both the above extensions have followed the spirit of our "optimistic" protocol. This is because neither of these protocols is as fault-tolerant as the protocols we presented in Chapter 3 and the added fault-tolerance of our conservative solution would not be useful. Solutions along these lines would be correct, however.

4.5 An example: point-to-point messages

We conclude this chapter with an example of how the ideas presented here can be used to integrate another common communication paradigm into the framework of virtual synchrony with causal message delivery. In many situations, messages do not need to be multicast to every member of a process group; a point-to-point message is sufficient. We would, however, like to maintain causal ordering between these point-to-point messages and messages multicast to the group.

Using the techniques of this chapter, we simply define $O(n^2)$ process groups of size 2, where n is the number of processes in the group — one process group for every pair of communicating processes. In order to keep the number of external timestamps low, each group of size 2 uses the conservative scheme for external timestamping. In practice, this means that after sending a point-to-point message, a process must wait for an acknowledgement to that message before sending a message to the process group as a whole (or any other process group). This delay is regrettable but unavoidable. Were we to use the alternative solution of using vector timestamps for each group, not only would we potentially have to maintain a large number of timestamps for each group, we would also have to ensure that all the point-to-point messages became stable with very high probability. This would be very expensive for point-to-point messages. The argument in favor of our method is that a single point-to-point RPC is fast and the cost is unaffected by the size of the system. Although one can devise more complex methods that eliminate the period of inhibited multicasting, problems of fault-tolerance render them less desirable.

4.6 Client-Server groups

We now consider the special case of fault-tolerant client-server process groups. In these groups, a small number of servers communicate with a large number of clients. Typically, a client will multicast a request to the servers, and one of the servers will multicast a reply to the client and the other servers. Although a large number of clients may be active over a long period of time, over a short period of time it is probable that the number of active clients will be small. Client-server groups raise two issues — first of all, the handling of membership changes, which we will not address here as it is addressed in [BCG91], and the total size of timestamps in the system. We now address this issue.

A straightforward implementation of client-server groups would simply use $c+1$, overlapping groups, where a group containing a client and the servers is created for each client, as well as group containing just the servers. However, this would result in very large numbers of external timestamps being created in the system — for example, if external timestamps were simply vector timestamps, the total size of all external timestamps for the client-server "group" would be $O(sc)$, where s is the number of servers and c is the number of clients. We would like to do better than this. There are several optimizations that allow us to do so.

4.6.1 Extending message stability

In the straightforward multiple-group implementation, any client or server that obtained an external timestamp representing a message sent to the servers by another client would automatically piggyback that timestamp on outgoing messages. In Section 3.3.3, we introduced the notion of message *stability* (a message is stable if it has been received at all its destinations) and used this to compress timestamps in a single process group. In a similar fashion, we extend the basic multiple-group algorithm to propagate message stability from group to another, *if and only if* the two groups represent clients of the same set of servers. When a client c holding an external timestamp t for a group that represents another client c' knows that all messages contained in t are stable, that external timestamp can be discarded, even if it has never been sent anywhere. So, at any time, a client will only hold external timestamps relating to non-terminated messages sent to or by the servers. When the client needs to communicate outside the entire client server group, it can either piggyback all the external client timestamps it currently holds, or it can wait until this number falls below some predetermined limit.

This scheme has several advantages. It maintains causal correctness, while reducing arbitrarily the size of an external timestamp for the entire client server group. It does however leave the servers with a considerable amount of work to do, as they must check all the external timestamps that may be piggybacked on any client request.

4.6.2 Modified Vector Times

In order to reduce the amount of work done at each client and server to maintain causality *inside* the client/server group, we introduce a new vector timestamp that contains one entry for each client and one entry for each server. This vector timestamp will count the total number of messages that have been multicast by each process. A client entry will count the number of multicasts that client has made to the servers. The first entry for each server will count the total number of messages that server has multicast to the other servers, including those that were also sent to clients. The timestamp is maintained and updated using the usual rules for vector timestamps. Clients simply use a sequence number to match up requests and replies, ignoring the timestamp. However, when a server s receives a message from another server, first of all it waits until it has received all the client messages indicated by the timestamp. If the vector timestamp entry for server s_i has the value t_i, this indicates that the first t_i messages sent by s_i are in the causal past of this timestamp. But the channels between processes are FIFO, so if s waits until it has received at least t_i messages in all from s_i, it will have received the messages that are in the causal past of the vector timestamp. At this point, delivery of the message that contained the causal timestamp can proceed.

4.7 Multiple-group total order

We now discuss how to extend the single-group total ordered multicast developed in Section 3.4 to provide a global total order on all multicasts in the system. Recall that the single-group protocol operates by assigning a unique priority to each message. This priority is assigned by a subset of processes in the group, known as the *ordering processes*. Messages are then delivered at every process in priority order. Priorities must be strictly increasing, but there is no requirement that they be dense. This basic protocol will not provide pairwise or global total order, since the priorities are assigned and message delivery is controlled on a per-group basis. There is no control over the order in which processes received messages from *different* groups to which they belong. Thus, the total message orderings associated with each group cannot be interleaved to form a pairwise or globally consistent total message ordering.

4.7.1 An example

Cooper [Coo90] has pointed out an instructive example where a globally consistent message ordering is required. Consider the classic "Dining Philosophers" problem [Dij72], which captures the essence of many resource allocation problems in distributed systems. In this problem, five philosophers are seated around a dinner table. There is a fork between every pair of philosophers. When a philosopher wishes to eat, he picks up the fork to one side, then picks up the fork to the other side, and starts eating. When he is finished eating, he replaces the forks in order that his neighbours may use them. The problem is that the philosophers may "deadlock" — there is a state of the system where no philosopher may make progress because they are each waiting for one of their neighbours to replace a fork, which may never happen.

Now consider solving this problem with overlapping process groups. There is a process for each fork and a process group for each philosopher; thus, two process groups overlap if the associated philosophers share a fork. Every philosopher who wishes to pick up a fork sends a request ABCAST in the associated process group. Upon receiving such an ABCAST, the fork process will grant the fork (with a CBCAST) if it is not in use; otherwise, the request is held. When a philosopher finishes eating, the associated fork processes are notified with CBCAST messages. Waiting requests are satisfied in FIFO order.

Now, if ABCAST messages are locally ordered inside a single process group, the above algorithm can easily lead to deadlock. Pairwise ordering is irrelevant, since deadlock can arise even when there is only one process in the intersection between groups. However, if all ABCAST messages are delivered in an order that is consistent with some global total order, deadlock will *not* arise. This is because a philosopher p_1 can only be waiting on philosopher p_2 if p_2's request message precedes p_1's request message in the global total order. So no philosopher can wait on himself, and deadlock cannot occur.

4.7.2 The algorithm

To extend our single-group ABCAST algorithm to give a global order, we first observe that the priority values assigned by the algorithm to messages in each group, can in fact be *globally* ordered, since they consist of a Lamport clock value extended to a total by a unique attribute of the process that sent the message — for example, the name of the process. If we can ensure that messages are delivered at every process in priority value order, no matter what group they were originally sent in, then there will clearly be a global ordering on all messages in the system, consistent with the order of message deliveries in each individual group. Thus the global delivery order will satisfy Condition 2.5. So we wish to ensure:

- Whenever a process p that is a member of more than one group delivers any message m with priority t, it has already delivered any message m' intended for it with priority $t' < t$, irrespective of the groups in which m and m' were sent.

In order to ensure this property, we modify the ABCAST algorithm as follows.

- We ensure that in every view for group g, for every group g' that overlaps with g, the set of *ordering processes* for g contains a process p that is a member of both g and g', and is also an ordering process for g'. This property is easy to ensure whenever the membership of g changes — as part of the flush algorithm, processes can exchange current membership information. Furthermore, if a process p that is *not* in the set of ordering processes for g needs to join (because it has joined some other group g'), it can simply perform a flush on p announcing that it has joined the ordering set. It suspends *any* ABCAST in g' until the flush of g completes. It can do this because it will be already be a member of the ordering set for g'.

- When proposing a priority for a message m sent in group g, an ordering process will wait until it has received the final priority for any message sent in any other group g'.[3]

[3]This wait can actually be avoided by introducing additional ordering processes, but so doing could increase the cost of the protocol.

- Instead of maintaining an ABCAST delivery queue for each group, the algorithm maintains a unified delivery queue for all groups of which the process is a member.

These modifications are sufficient to ensure the correctness of the algorithm. To see this, consider a process p that is a member of two groups g and g'. Let p_o be an ordering process that is also a member of both g and g'. Suppose, by way of contradiction, that p delivers m with priority t_m and later delivers m' with priority $t_{m'} < t_m$, where m was sent in g and m' was sent in g'. Since $t_{m'} < t_m$, p_o proposed an ordering for m *after* it received the final ordering for m'. In that case, the final ordering message for m will be causally after the initial ordering request for m', and $m' \in PC_m$ at every process that will receive both, including p. But if $m' \in PC_m$ and $t_{m'} < t_m$, then m' will be delivered before m, contradicting our initial assumption. Note that causality must be maintained across multiple groups for this to work.

The cost of running the algorithm above in a given group g is proportional to the size of the ordering set. This, in turn, will be roughly proportional to the number of of groups with which g overlaps. Although it is possible to give algorithms that provide pairwise total order without providing global total order, these algorithms will also have message costs of the same order.

4.8 Conclusion

This chapter has given a general framework for constructing algorithms for maintaining causality in overlapping process groups. The basic idea is to encapsulate the causal obligations of a process group inside a single "external timestamp" that is piggybacked on messages outside that process group. This framework has been illustrated by deriving two distinct causality algorithms, differing mainly in their choice of external timestamp. We also used it to extend causality maintenance to include point-to-point messages within a single process group, and showed how it could be used to extend other single-group algorithms in the literature. Finally, we developed an algorithm for maintaining a global total order across a set of multiple overlapping process groups. In the next chapter, we will examine the issue of external timestamps more closely; in particular, we will look at methods for reducing the number of external timestamps that must be maintained in the system.

To send a message m to process group g:

1. Perform a *tstamp* operation on every group that the client has interacted with *since* it last sent a message to g. Set the external timestamp for each local group to be the value returned. If the new value is greater than the old value, clear the *forwarded* list for that group.
2. Append to m a copy of every external timestamp in the list that has not already been forwarded to g, according to the *forwarded* lists for each message. Update the *forwarded* lists accordingly.
3. Wait until all previous messages that the client has previously sent to g have been sent.
4. Send the message to g, using the single-group algorithm for g.

On receipt of a message m from a process group g:

1. For each external timestamp $t_{g'}$ for some local group g' appended to the message:
   - If $t_{g'}$ is not less than or equal to $T_{g'}$, the current value of the external timestamp for g', perform $synchronize_p(g', t_{g'})$. Update $T_{g'}$ to be $merge(T_{g'}, t_{g'})$.
2. Wait for all the *synchronize* operations to complete.
3. For each external timestamp $t_{g'}$ for some non-local group g' appended to the message:
   - Update $T_{g'}$, the current value of the external timestamp for g', to be $merge(T_{g'}, t_{g'})$. If the new value is greater than the old one, clear the *forwarded* list.
4. Wait for all previous messages received from g to be delivered to the client. Then deliver m to the client and call $delivered(g, m)$.

Figure 4.3: The general multiple-group algorithm

Chapter 5

External Timestamps Reconsidered

5.1 Introduction

The previous chapter gave a general algorithm for maintaining causality across multiple groups, using the concept of *external timestamps.* In that algorithm, each process group potentially had to maintain and transmit external timestamps corresponding to every other group in the system. In this chapter, we look at methods for reducing this cost, and examine the situations under which one process group g_1 may ignore the external timestamps corresponding to another process group g_2. This issue critically depends on the potential communication paths between the two groups — the set of groups through which causal chains relating messages in the two groups could form. Clearly, new communication paths can be formed and old ones eliminated over the execution system, as processes join groups, leave groups, and fail. Section 5.2 gives a graph-based model, the *communication structure,* that we will use to model the potential communication paths in the system. Although the topology of communication structure can vary dynamically, we will develop our algorithm to handle this by first dealing with the static case in Section 5.3, where we derive an optimal algorithm for that case. Section 5.4 then extends this algorithm into a general algorithm for handling the dynamic case. Finally, Section 5.5 shows how we can use the algorithm we have just developed for periodically garbage-collecting timestamps even if topology changes have not occurred.

5.2 Communication Structure

Define the *communication structure* of a system to be an undirected graph $CG = (G, E)$ where the nodes, G, correspond to process groups and edge (g_1, g_2) belongs to E iff there exists a process p belonging to both g_1 and g_2. If the graph so obtained has no biconnected component [1] containing more than k nodes, we will say that the communication structure of the system is k-bounded. In a k-bounded communication structure, the length of the largest simple cycle [2] is k. A 0-bounded communication structure is a tree (we neglect the uninteresting case of a forest). Clearly, such a communication structure is acyclic.

Notice that causal communication cycles can arise even if CG is acyclic. However, the acyclic structure *restricts* such communication cycles in a useful way – such cycles will either be simple cycles of length 2, or complex cycles. This is because any cycle involving more than two processes must include the same edge more than once. This is a direct consequence of the graph acyclicity.

The algorithms we give below will not explicitly construct or maintain the communication structure. They will, however, *reason* about it, and it will be used in our proofs. Thus, our use of the communication structure is akin to the use of deadlock detection graphs in more traditional operating systems settings [BHG87].

[1] Two vertices are in the same biconnected component of a graph if there is a path between them after any other vertex has been removed.

[2] The nodes of a simple cycle (other than the starting node) are distinct; a complex cycle may contain arbitrary repeated nodes.

5.3 Static Communication Structures

Below, we demonstrate that it is unnecessary to transport all changed vector timestamps on each message when the communication structure is static and known in advance. The algorithm being discussed here is not given in full, as it is a straightforward modification to the general purpose multiple group algorithm (Figure 4.3); we are simply giving extra rules for deciding what timestamps need to be kept on the external timestamp list. Consider a process group g_1, that is a member of biconnected components $C_1, C_2 ... C_n$ of the communication structure. We show that each process within this group need only:

1. Maintain the external timestamps of other groups in $C_1 ... C_n$.
2. Need only transmit them on messages being multicast to groups in the appropriate biconnected components. We can also show that they need to maintain *at least* these timestamps.

For example, if g_i is a member of biconnected component C_i, then if the external timestamp maintained for g_i by g_1 changes, then g_1 must put the new timestamp on the next message it multicasts to any group in C_i. As a consequence, if the communication structure is acyclic, processes need only maintain the timestamps for the groups to which they belong.

We proceed to the proof of our main result in stages. First we address the special case of an acyclic communication structure. In this case, each process runs the general multiple-group algorithm of the previous chapter, modified to maintain on the external timestamp list only the external timestamps of groups of which it is a member. We will refer to this algorithm as the *acyclic multiple group algorithm.* The proof of liveness for the general multiple-group algorithm (Theorem 4.2) clearly also applies in this case; it remains to prove safety — i.e., that the acyclic multiple-group algorithm does in fact deliver messages in causal order.

Lemma 5.1 *If a system has an acyclic communication structure, the acyclic multiple group algorithm delivers messages in causal order.*

Notice that under this lemma, the overhead on a message is limited by the size and number of groups to which a process belongs.

We wish to show that if message m_1 is sent (causally) before message m_k, then m_1 will be delivered before m_k at all overlapping sites. Consider the chain of messages below.

$$p_1 \overset{\mathrm{m}_1}{\underset{\mathrm{g}_1}{\rightsquigarrow}} p_2 \overset{\mathrm{m}_2}{\underset{\mathrm{g}_2}{\rightsquigarrow}} p_3 \overset{\mathrm{m}_3}{\underset{\mathrm{g}_3}{\rightsquigarrow}} \ldots\, p_{k-1} \overset{\mathrm{m}_{k-1}}{\underset{\mathrm{g}_{k-1}}{\rightsquigarrow}} p_k \overset{\mathrm{m}_k}{\underset{\mathrm{g}_k}{\rightsquigarrow}} p_{k+1} \tag{5.1}$$

In words, p_i sends message m_i in group g_i, p_{i+1} received m_i and then sends m_{i+1} in group g_{i+1}, and so on. Our proof will show that if $m_i \rightarrow m_j$ and the destinations of m_i and m_j overlap, then $m_i \overset{\mathrm{p_j}}{\rightarrow} m_j$, where p_j is the sender of m_j.

We now note some simple facts about this message chain that we will use in the proof. Recall that a multicast to a group g_a can only be performed by a process p_i belonging to g_a. Also, since the communication structure is acyclic, processes can be members of at most two groups. Since m_k and m_1 have overlapping destinations, and p_2, the destination of m_1, is a member of g_1 and of g_2, then g_k, the destination of the final multicast, is either g_1 or g_2. Since CG is acyclic, the message chain $m_1 ... m_k$ simply traverses part of a tree, reversing itself at one or more distinguished groups. An example is given in figure 5.1, where the message chain $m_1 ... m_4$ reverses itself at g_4. We will denote such a group g_r. Although causality information is lost as a message chain traverses the tree, we will show that when the chain reverses itself at some group g_r, the relevant information will be "recovered" on the way back.

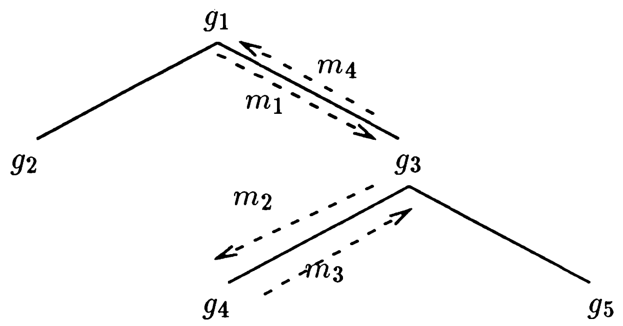

Figure 5.1: Message chain reversal

Proof: The proof is by induction on l, the length of the message chain $m_1...m_k$. Recall that we must show that if m_1 and m_k have overlapping destinations, they will be delivered in causal order at all such destinations; i.e., m_1 will be delivered before m_k.

Base case. $l = 2$. Here, causal delivery is trivially achieved, since $p_k \equiv p_2$ must be a member of g_1 and m_k will be transmitted with g_1's timestamp. It therefore will be delivered correctly at any overlapping destinations.

Inductive step. Suppose that our algorithm delivers all pairs of causally related messages correctly if there is a message chain between them of length $l < k$. We show that causality is not violated for message chains where $l = k$.

Consider a point in the causal chain where it reverses itself. We represent this by $m_{r-1} \rightarrow m_r \rightarrow m_{r'} \rightarrow m_{r+1}$, where m_{r-1} and m_{r+1} are sent in $g_{r-1} \equiv g_{r+1}$ by p_r and p_{r+1} respectively, and m_r and $m_{r'}$ are sent in g_r by p_r and $p_{r'}$. Note that p_r and p_{r+1} are members of both groups. This is illustrated in Figure 5.2. Now, $m_{r'}$ will not be delivered at p_{r+1} until m_r has been delivered there, since they are both multicast in G_r and from the properties of the *tstamp* and *sync* operations. We now have $m_{r-1} \stackrel{p_r}{\rightarrow} m_r \stackrel{p_{r+1}}{\rightarrow} m_{r+1}$. We have

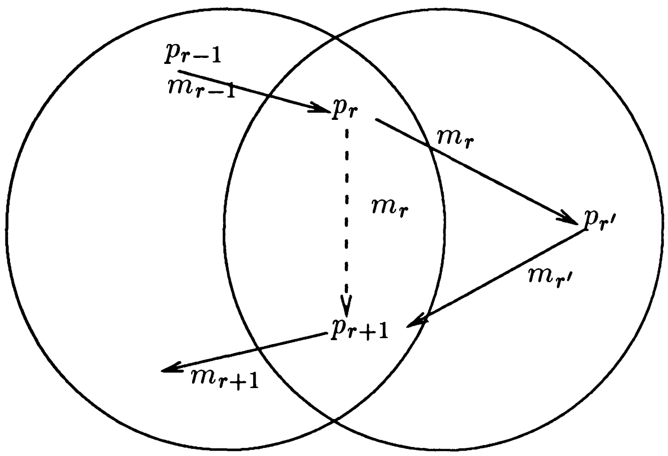

Figure 5.2: Causal Reversal

now established a message chain between m_l and m_k where $l < k$. So, by the induction hypothesis, m_l will be delivered before m_k at any overlapping destinations, which is what we set out to prove.

□

We now proceed to consider a more general algorithm – the *biconnected multiple-group algorithm.* In this algorithm, each process runs the multiple-group algorithm, modified to maintain on its external list the external timestamps of each group in the biconnected components of CG to which it belongs. Furthermore, an external timestamp for a group g in a biconnected component C_g is only piggybacked on messages multicast to other groups in C_g.

Theorem 5.1 *The biconnected multiple-group algorithm always delivers messages in causal order.*

Proof: As with Lemma 5.1, our proof will focus on the message chain that established a causal link between the sending of two messages with overlapping destinations. This sequence may contain simple cycles of length up to k, where k is the size of the largest biconnected component of CG. Consider the simple cycle illustrated below, contained in some arbitrary message chain.

$$p_1 \overset{m_1}{\underset{g_1}{\rightsquigarrow}} \ldots p_2 \overset{m_c}{\underset{g_c}{\rightsquigarrow}} p_3 \overset{m_{c+1}}{\underset{g_1}{\rightsquigarrow}} \tag{5.2}$$

Now, since p_1, p_2 and p_3 are all in groups in a simple cycle of CG, all the groups are in the same biconnected component of CG, and all processes on the message chain will maintain and transmit the timestamps of all the groups. In particular, when m_c arrives at p_3, it will carry a copy of the external timestamp for g_1 that is causally subsequent to m_1, or such an external timestamp will have previously arrived. This means that m_c will not be delivered at p_3 until m_1 has been delivered there. So m_{c+1} will not be transmitted by p_3 until m_1 has been delivered there. Thus $m_1 \xrightarrow{p_3} m_{c+1}$. We may repeat this process for each simple cycle of length greater than 2 in the causal chain, reducing it to a chain within one group. Thus, we can now show that this execution of the biconnected multiple-group algorithm is isomorphic to an execution of the acyclic multiple-group algorithm. We now apply Lemma 5.1, completing the proof. □

Theorem 5.1 shows us what timestamps are sufficient in order to assure correct delivery of messages. Are all these timestamps in fact necessary? Yes. It is easy to show that if a process that is a member of a group within a biconnected component of CG does not maintain an external timestamp for some other group in CG, causality may be violated. We therefore state without formal proof:

Theorem 5.2 *If a static system uses any variant of the general multiple-group protocol to maintain causality, it is both necessary and sufficient for a process p_i to* *maintain and transmit those external timestamps corresponding to groups in the biconnected component of CG to which p_i belongs.*

5.4 Dynamic communication structures

We now address the issue of dynamic communication structures. This raises two main points:

- How do we discover topological information in a dynamic system?
- How do we detect and remove invalid information after the topology has changed?

We address these points by introducing first of all a *global Lamport clock*, a single integer maintained by every process and piggybacked on every message in the system. Secondly, we introduce the notion of a background topology algorithm that is used to find "safe" topological information about the system. Our general dynamic algorithm will run a background topology algorithm to trim the external timestamp list. If the topology of the system changes, the old results of the background topology algorithm will be discarded, and a new run will be started. Since it is possible that causality will be violated during the changeover, the global Lamport clock is used at this time to ensure correct message delivery.

5.4.1 The global Lamport clock

Each process p maintains its copy of LC, the global Lamport clock as follows:

- p may increment LC_p, its local copy of LC, at any time.
- Upon receiving a message containing a Lamport clock value $LC_m > LC_p$, p performs the following operations *before* performing any other operations with m:

  - p performs and completes a flush operation on every group of which it is a member. The value LC_m is distributed with the flush operation.
  - p sets $LC_p = LC_m$.

- Every process p' that participates in the flush operation sets $LC_{p'} = LC_m$ at the end of the flush operation.

Intuitively, the purpose of the local clock is to force delivery of all messages that *might* be causally before the latest message received. This is formalized in the following lemma:

Lemma 5.2 *If $LC_{m_1} < LC_{m_2}$, then m_1 will be delivered before m_2 at all overlapping destinations.*

Proof: Suppose m_1 was multicast in g_1 and m_2 was multicast in g_2. If g_2 does not overlap with g_1, the lemma is trivially true. So suppose g_1 and g_2 overlap. Before m_2 is delivered to any process that is a member of g_1, that process must perform a flush operation on g_1 carrying the value LC_m. Since $LC_{m_1} < LC_{m_2}$, m_1 must be causally ordered before the flush. But by the properties of flush, m_1 will be received and delivered before the completion of the flush. Thus m_1 will be received and delivered before m_2. □

The global Lamport clock is quite similar to the LT clock introduced in Chapter 3; it is updated in an analagous fashion, and the group flush used here is analogous to the channel flush used there.

5.4.2 Background Topology Algorithms

The background topology algorithms we use need only work in *static* or *limited dynamic* communication structures. They have the opportunity to give up if the situation gets too complicated! They work in the background at each process, and may piggyback information on other outgoing messages or may generate new messages.

A background topology algorithm a running at a process p that is a member of a group g extends and interfaces with the multiple-group algorithm by providing a *potential external timestamp list* — the list of groups whose timestamps should be maintained on the external timestamp list should such timestamps arrive. By theorem 5.2, the potential external timestamp list must contain at least every group that is in the same biconnected component of the communication structure as p. Such a potential external timestamp list is *valid*; otherwise it is *invalid*. More precisely, the interface is as follows:

- a can observe every group g' that g is adjacent to, before p receives any message causally after a message in g'.

- a may provide a list of groups whose timestamps need to be kept on the external timestamp list at p.

- a must indicate to p when any external timestamp list that it has provided at any process p' has become invalid. Note that a is free to provide this indication if invalidation *might* have occurred. The indication must be provided before any subsequent message is sent.

The first point can be implemented at group membership change time, by having each process lazily multicast the membership change in *every* group of which it is a member before multicasting any client messages that observed the change. The other points can be implemented locally.

The algorithm of Figure 5.3 is a simple example of a background topology algorithm. This algorithm tries to find trees in the communication structure. It is easy to see that step 2 ensures that in a static communication structure, all the nodes that are contained in subtrees will become green. Step 3 ensures that if a subtree becomes a biconnected component (e.g., two distinct leaves form an edge between them) at least one node notices and can "raise the alarm".

1. Every process group starts coloured red.
2. A process group that is adjacent to at most one other red group, colours itself green and sets the potential external timestamp list to be the immediate neighbours of the process group in the communication structure.
3. A green process group that *becomes* adjacent to any other process group invalidates the potential external timestamp list.

Figure 5.3: Tree finding topology algorithm

5.4.3 The general dynamic algorithm

We are now in a position to bring the previous two subsections together and give the general dynamic algorithm. The basic idea is to use the global Lamport clock to divide the execution of the system into "epochs". A separate instance of the background topology algorithm is executed in each epoch. As it executes, this algorithm will note certain facts about the current topological state of the system, and based on this information, will provide each process in the system with potential timestamp lists. If the topology changes in such a way that one of the potential timestamp lists may be incorrect, a new epoch is started and a new instance of the topology algorithm runs in this epoch. When any process detects a new epoch, before interacting with any process group of which it is a member, it performs a flush operation on that process group. This flush will satisfy all causal obligations related to the old epoch; therefore once it commences at any process, that process can dispose of all external timestamps that it holds relating to the old epoch. When a process detects a new epoch, it throws away its potential external timestamp list and keeps all timestamps on received messages until a new potential external timestamp list is provided. The detailed algorithm is given in figure 5.4.

1. Initialize the local epoch counter to 0. Piggyback this epoch counter on all outgoing messages.
2. Mark the potential timestamp list as invalid.
3. Start running a local instance of the background topology algorithm.
4. Whenever the background topology algorithm delivers a new valid external timestamp list, stop maintaining and transmitting any timestamps not on the list.
5. Whenever a message is received with an epoch counter higher than the current local epoch counter:
   - Perform a flush on the group from which the message was received.
   - Set the local epoch counter to the value received.
   - Invalidate the timestamp list
   - Deliver the message.
6. Whenever the potential timestamp list becomes invalid, increment the epoch counter, start maintaining all external timestamps received, and transmitting any external timestamps that change, as in the basic algorithm of the previous chapter.

Figure 5.4: Dynamic topological timestamp reduction

Theorem 5.3 *(Safety) The dynamic timestamp reduction algorithm always delivers messages in causal order.*

Proof: Suppose not. The only reason for delivering a message m out of causal order is that the message did not follow or contain a necessary external timestamp t. Clearly, the generation of the external timestamp and delivery of the message must have happened in the same epoch, since otherwise any message contained in the external timestamp would have been delivered in the flush induced by the change of epoch step 5. Therefore no process on the message chain between the generation of t and the delivery of m had its external timestamp list invalidated by the background topology algorithm. But the background topology algorithm *must* invalidate the external timestamp list at a process if there is is a possibility that p could send a message without a necessary external timestamp. But if such an invalidation had occurred, the epoch variable would have changed. □

Theorem 5.4 *(Liveness) The dynamic timestamp reduction algorithm always delivers every message sent.*

Proof: Since the algorithm does not delay messages except in the flush step, which will always terminate provided that the flush terminates, it will always deliver any message sent. □

5.5 Garbage-collecting timestamps

The epochs of the previous section can also be used to garbage-collect timestamps even if the communication topology is not changing. It is easy to see that if any process wishes to dispose of its pool of external timestamps, all it needs to do is start a new epoch. This operation is clearly not without cost, as each process group will perform a FLUSH as it detects the new epoch. But this algorithm could be useful in situations where timestamps occasionally take up many system resources, and must be disposed of.

5.6 Conclusion

This chapter has closely examined the problem of minimizing the number of external timestamps that must be maintained when using the general-purpose multiple group causality algorithm. We have derived an optimal algorithm for the static case, and a family of algorithms for the dynamic case that trade off the number of timestamps that can be eliminated with the number of extra background messages in the system. Finally, we have shown how these algorithms can also be used to garbage-collect timestamps.

Chapter 6

Performance and Transport Protocols

In this chapter, we discuss the initial implementation and performance of our protocol. We show that the performance of our protocol will be largely dominated by the performance of the underlying layer that is simply concerned with moving data from one site to multiple other sites. We then discuss the design of some alternatives for this layer.

6.1 Complexity and overhead of the protocol

Our protocols have an overhead of both space and messages transmitted. Within a single group, the size of a message will be increased by the size of its vector timestamp. Vector timestamp sizes for various types of groups are summarized in Table 6.1.

This protocol works well with all the styles of process group usage considered in Section 1. In peer groups and diffusion groups, one vector timestamp is piggybacked on each message. In the case of diffusion groups, the number of entries in the timestamp can be optimized —instead of an entry for each member of the group,

Table 6.1: Timestamp sizes resulting from different process group styles

Peer Groups	$k, k \leq n$ n is the number of group members.
Client/Server Groups	$k, k \leq s + c$ s is the number of servers and c is the number of clients.
Diffusion Groups	$k, k \leq b$ b is the number of members multicasting into the group.
Hierarchical Groups	$k, k \leq n$ n is the number of members in any group.

there need only be an entry for each *sending process* in the group. Hierarchical groups fall naturally into the acyclic VT scheme, and the total size of client/server group timestamps can be kept reasonable using the methods of Section 4.6.

The size of a message will also be increased by the number of external timestamps that it carries; we expect that most external timestamps carried by a message will be for groups that are relatively near to it in the communication structure graph, and will be small. This is because an external timestamp will only be piggybacked when it has changed; changes will not be often observed for groups that are far away, due to locality of communication.

The number of overhead messages sent will depend on the number of non-piggybacked messages sent to propagate message stability information, and, when implemented, the frequency of group flushing induced by epoch changes. In ISIS, epoch changes are expected to be rare and terminate messages are always piggybacked on a subsequent CBCAST unless communication in a group quiesces.

We believe that latency, especially when the sender of a multicast must delay before continuing computation, is the most critical and yet unappreciated form of overhead. Delays of this form are extremely noticeable. In many systems, there is only one active computation at a given instant in time, or a single computation that holds a lock or other critical resource. Delaying the sender of a multicast may thus have the effect of shutting down the entire system. In contrast, the delay between when a message is sent and when it reaches a *remote* destination is less relevant to performance. The sender may be delayed in two ways: if the transmission protocol itself is computationally costly, or if a self-addressed multicast cannot be delivered promptly because it is unsafe to do so. Defined in this sense, our method imposes latency on the sender of a causal multicast only in the conservative protocol, when a process needs to communicate in one group after communicating in another. Otherwise, the protocol is totally asynchronous. Some latency in a totally ordered multicast is unavoidable. Our scheme for using a set of "ordering processes" allows the system designer to tune the latency according to the exact semantics required.

6.2 Implementation

The ISIS group has implemented our CBCAST and a version of ABCAST[1] within the ISIS toolkit. The initial ISIS CBCAST implementation consists of the VT scheme and the conservative rule, together with the view–flush and message stability protocols. The protocol server described in [BJ87b] is used as a source of group membership views. The multicast transport protocol does *not* take advantage of any special multicast transport hardware — it is simply layered on top of UDP, an unreliable point-to-point datagram protocol. The basic ISIS multicast transport protocol is designed around a point-to-point model. Each process in a group maintains a two-way reliable data stream with each other process in the group. Whenever possible, acknowledgement information is piggybacked on other packets, such as replies to an RPC or multicast. These streams are maintained independently of each other; for brevity, we omit discussion of such details as flow control and failure detection. This scheme has several advantages; first, it is relatively easy to understand, as it is based on a well-known communication model. Also, since it is built on top of unreliable datagrams, it can be easily implemented on any network that provides this service. It has, however, several disadvantages — in particular, it does not

[1] limited to a single ordering process

scale well. As we will see below, the processing and network transmission costs of communicating with a group rise linearly with the number of processes in the group. We have therefore investigated the design of other multicast transport protocols, discussed in Section 6.4.

6.3 Measured Performance

We measured the cost of sending RPCs to one or more remote sites and the cost streaming causal messages to one or more remote sites. The RPC request was sent in a CBCAST; the result returned in a CBCAST reply packet. A new lightweight task was created at the receiver to field each RPC request. An ISIS message is fairly complex and allows scatter/gather and arbitrary user-defined and system-checked types. Since no attempt has been made to optimize message data structures for the simple case of a null RPC, this accounts for a large part of the time spent in the messaging/task layer of the system.

Table 6.2 shows that the cost of transmitting to a group grows roughly linearly with the size of the group. The actual figures are comparable to what those achievable using the vendor-supplied remote procedure mechanism for the machines with which we worked. For example, a remote procedure call using a 1kb message through our facility (with a null reply) had a round-trip latency of 8.79ms. The figure for SUN RPC is comparable. Similarly, the throughput figures compare quite well with traditional streaming protocols such as TCP (we obtained 673 null messages per second, or 600kb of data per second). A graph of this data appears as Figure 6.1.

In order to understand how CPU time was expended by our protocol, we profiled the protocol within a single processor and between a pair of processors using RPC over a 10Mbit Ethernet. We then computed the costs attributable to different parts of the system. Table 6.3 shows a profile for a null RPC sent by a thread to another entry point within its own address space. This involves creating a new thread to

Table 6.2: Multicast performance figures.

S: null packets M: 1K packets; L: 7K packets

All figures measured on Sun 4/60's running SUNOS 4.0.3

Group Size	Causal multicast with replies (ms) S	M	L	Asynchronous msg/sec S	kb/sec L
1	1.42	1.46	1.44	2469	16706
2	7.36	8.79	18.3	673	601
3	9.66	11.1	26.6	361	440
4	12.0	15.1	35.4	288	282
5	16.4	17.8	44.3	224	203
6	17.8	21.1	55.4	215	151
7	21.5	26.2	60.4	177	125
8	25.2	30.1	66.6	150	104

Table 6.3: Null causal multicast to another local thread with reply

Operation	Total Cost	Causal Cost
send	406 uS	86 uS
receive	394 uS	31 uS
reply	406 uS	86 uS
receive	224 uS	31 uS
Total	1.43 mS	234 uS
(Measured)	1.43 mS	

handle each delivered message but no communication outside of the address space of the test program. The first column shows total (measured) costs and the second shows costs attributed to causality, which we obtained by comparing the costs of sending causally ordered and FIFO messages.

Table 6.4 estimates the costs on a layer-by-layer basis for the 8.79ms RPC to a remote process. The layer costs were based on measurements using static test programs (loops that exercised some component of the protocol layer). The costs are broken down into the time spent in the protocol implementation (taken from

Table 6.4: 1k causal multicast to a remote process with reply

Isis Layer	1.43 mS
Transport	3.6 mS
System Calls	1.59 mS
Wire Time	1.8 mS
Total	8.42 mS
(Measured)	8.79 mS

the null multicast table, above), the transport costs, costs spent in system calls, and the time on the wire for a 1k message with its ISIS-supplied header and a null reply. The header size used was approximately 400 bytes in each case.

Our first major conclusion is that we have succeeded in significantly improving ther performance of ISIS; even allowing for changes in CPU speed, these numbers are much better than those reported in [BJ87a]. Our second major conclusion we draw from these performance studies is that nearly all the time spent in our new protocol is in the layers concerned with physically transporting messages to a remote machine. Our protocol imposes little cost in relation to this number. This shows us that exploitation of hardware multicast will be important. The next section outlines some considerations for designing a multicast transport protocol for use by ISIS.

6.4 Multicast Transport

As shown in the previous section, an improved multicast transport protocol is vital for improving the performance of our protocol. For our system an ideal multicast transport protocol would have the following features:

- It would be independent of network topology, but able to take advantage of features of particular networks — e.g., a broadcast subnet.

- The cost of sending a message would be independent of the number of recipients of that message.

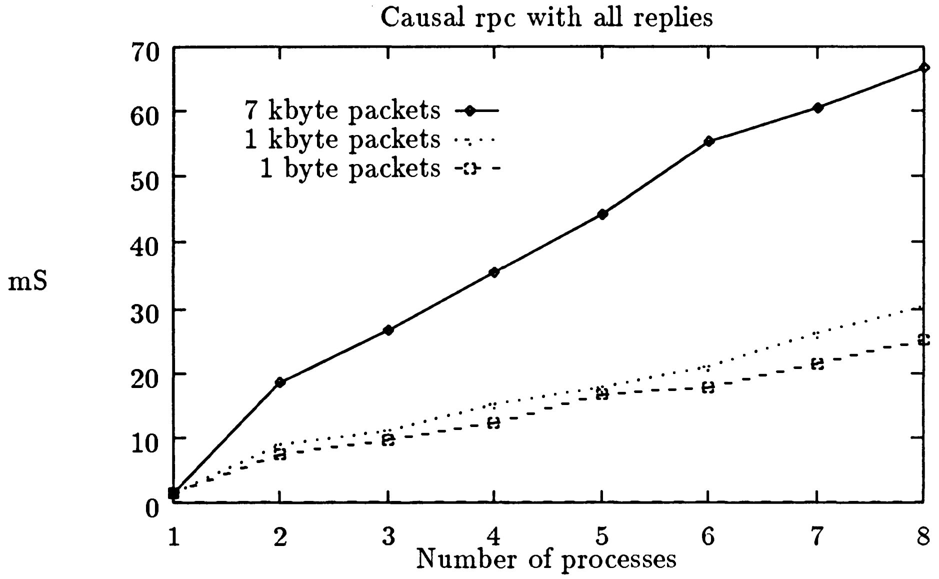

Figure 6.1: RPC time as a function of message and group size

- It would work efficiently for both small and large messages.
- It would have low overhead, low latency and high throughput.

It is also important to note that frequently a multicast may give rise to many replies directed to the original sender. We call such an occurrence a *convergecast*. This can lead to congestion at the original multicast sender, with many of the replies being lost. To avoid this, a multicast transport protocol should have some sort of mechanism for co-ordinating and reliably delivering multicast replies. Similar considerations may apply to acknowledgements. However, acknowledgements need not be as timely as replies — the multicast transport protocol has more freedom to delay them.

Generally speaking, a reliable multicast transport mechanism will be used in two distinct modes. In the first mode, *stream* mode, one process will multicast a large amount of data to the group before another process wishes to multicast (or even reply). Multicasting is more or less continuous. For example, this usage could arise in a trading system, where the transport mechanism is being used to disseminate quotes to trading stations. Another example is a replicated file system where a client workstation is writing a file to a group of file servers. In the second mode, *rpc* mode, many processes multicast replicated rpc's to the group, where each rpc contains relatively little data, and may or may not actually require replies. Multicasts are not continuous, but bursty. This could arise in maintaining and querying a distributed database or maintaining the state of a distributed game. Note that the application using the multicast transport protocol can provide hints as to the mode that may be most appropriate. Intermediate modes of usage can of course arise; we do not expect them to be common.

Reliable multicast transport protocols may be divided into two classes: those based on positive acknowledgements, and those based on negative acknowledgements. Many previous proposals for reliable multicast transport protocols have been based on negative acknowledgements, including [KTHB89, AHL89, CM84, FM90]. (These protocols, in addition to providing reliable transport, also provide transport ordering properties.) This is because the designers of these protocols believed that a positive acknowledgement from each receiving site would be expensive. We do not believe that this is so.

If a process group is largely communicating in *rpc* mode, reply messages will be converging at the sender in any case. These reply messages can carry positive acknowledgements. In addition, if there are many of these reply messages, they should be scheduled by some mechanism to avoid congestion and message loss at the multicast sender. On the other hand, if a group is largely communicating in *stream* mode, the issue of flow control becomes very important. The sender can't send data faster than the slowest process in the group can receive it; in order to avoid packet loss, there will be flow control packets coming back to the sender from each of the other processes in the group. Again, these packets may carry positive acknowledgments, and again, they must be scheduled in order to avoid congestion problems. The protocol has more flexibility in scheduling these packets than in scheduling reply packets, since they do not contain data that needs to be delivered to the higher level.

There are several possible mechanisms for scheduling packets that are converging on the same destination. One scheme is for the original sender to schedule the packets; it will decide how many concurrent acknowledgments or replies it (and the network) can handle. It then schedules each group of acknowledgements. This scheme involves some extra work by the sender, but it has the advantage that the sender can control the rate at which the packets come back depending on whether or not his client is waiting for replies.

Other methods involve the receivers co-operating to ensure that they don't send too many packets to the sender at once. One such method basically involves passing one or several tokens around the group, with the holder of a token having the right to send reply or acknowledgement packets to the original sender. If the replies or acknowledgements are small, they can be put on the token itself, which is returned to the sender when it is full. The main problem with this scheme is that the acknowledgement or reply may take a long time to return to the original sender of a message. This can be overcome by using large window sizes, or by using a large enough number of tokens. Another problem is that the overhead of receiving a message is higher, because an acknowledgement token must be received and transmitted also. This can be solved by having one token acknowledge several messages, and by piggybacking the acknowledgement token wherever possible. A third problem is that the loss of one acknowledgement packet may cause a message to be retransmitted to multiple destinations. We believe that the extra overhead is acceptable, since packet loss should be rare.

Another receiver-scheduled method for handling acknowledgements or replies is simply to have each acknowledgement be returned at some random time by the recipients. This scheme has been extensively analyzed by [Dan89]; the main problem is that in order to avoid congestion at the original sender, the interval from which the random delays must be picked is very long. It is also of course possible to combine several of the above schemes. For example, acknowledgements could be sender-scheduled in small groups, and individual acknowledgements within each group could be further randomly delayed.

We are implementing multicast transport protocols with several of the convergecast avoidance scheduling strategies described above, and will experiment with them as alternatives to the basic ISIS multicast transport protocol. Our implementations are based on the multicast UDP software of [Dee88], which provides a logical unreliable multicast across internets independently of whether the underlying networks support physical multicast. Full details of the design and implementation of these protocols will be found in a future paper.

6.5 Conclusion

This chapter has considered the performance of the protocols outlined in the previous chapters. Based on a preliminary implementation, we have shown that the cost of our protocols will be dominated by the cost of moving data from one site to several other sites — in other words, a reliable multicast. We have outlined some considerations for the design of such a reliable multicast protocol.

Chapter 7

Discussion

We have developed a new suite of protocols for implementing virtual synchrony, an abstraction that is exceeding useful for programming distributed systems. We started in Chapter 2 by outlining our model an system architecture. In Chapter 3 we examined protocols for a *single* process group, and gave new causal ordering protocols, a new total ordering protocol, and a new group flush protocol. Both the total ordering protocol and the group flush protocol are based in part on the causal ordering protocol. Chapter 4 extended this work to give causal and total ordering protocols for multiple overlapping process groups. Our total ordering protocol can give a system-wide total ordering using only local communication. In Chapter 5, we examined optimizations to our multiple-group total ordering protocol, and showed that in the case of a static communication graph, our protocol is optimal. We also gave a family of protocols for dynamic communication graphs. Chapter 6 reported on a preliminary implementation of our protocols.

7.1 Related Work

There has been a great deal of work on multicast primitives. CBCAST-like primitives are described in [BJ87b,PBS89,VRB89,SES89,LLS90]. Our work is most closely related to that of Ladin and Peterson. Both of these efforts stopped at essentially the point we reached in Chapter 3 arriving at protocols that would perform well within a single small group, but subject to severe drawbacks in systems with large numbers of processes and overlapping, dynamically changing process groups. Pragmatic considerations stemming from our desire to use the protocol in ISIS motivated us to take our protocol considerably further. In fact, the methods that we have developed can equally apply to their protocols. Also, we believe the resulting work to be interesting from a theoretical perspective.

Viewed from a practical perspective, an ordered multicast protocol suite that scales well and imposes little overhead under typical conditions certainly represents a valuable advance.

ABCAST-like primitives are reported in [CM84,BJ87b,GMS89,PGM85]. Our ABCAST protocol is motivated by the Chang-Maxemchuck solution [CM84], but is simpler and faster because it can be expressed in terms of a virtually synchronous CBCAST. In particular, our protocol can be configured to avoid the potentially lengthy delays required by the Chang-Maxemchuck approach prior to committing a message delivery ordering. We believe this argues strongly for a separation of concerns. In particular, we derive a substantial benefit by decoupling process group management from the communication primitive itself.

We note that of the many protocols described in the literature, very few have been implemented, and many have potentially unbounded overhead or postulate knowledge about the system communication structure that might be complex to deduce. This makes direct performance comparisons difficult, since some published protocols give performance estimates based on simulations or measure dedicated implementations on bare hardware. We are confident that our new ISIS communication suite gives performance fully competitive with any alternative. The ability to extend the transport layer will enable the system to remain competitive even in settings with novel architectures or special communication hardware.

7.2 Future Work

The ability to run our protocols over new transport protocols raises some new questions for future investigation, in addition to those mentioned in Chapter 6. For example, one might run our CBCAST over a transport layer with known realtime properties. Depending on the nature of these properties, such a composed protocol could satisfy both sets of properties simultaneously, or could favor one over the other. The delay of flushing channels suggests that realtime and virtual synchrony properties are fundamentally incompatible, but this still leaves open the possibility of supporting a choice between weakening the realtime guarantees to ensure that the system will be virtually synchronous and weakening virtual synchrony to ensure that realtime deadlines are always respected. For many applications, such a choice could lead to an extremely effective, tuned solution. Pursuing this idea, we see the Isis system gradually evolving into a more modular structure composed of separable facilities for group view management, enforcing causality, transporting data, and so forth. For a particular setting, one would select just those facilities actually needed. Such a compositional programming style has been advocated by others, notably Larry Peterson in his research on the Psync system.

On the more theoretical side, the dynamic external timestamp algorithms could be further optimized in several fashions. We have only give simple background topology algorithms; much more could be done here. We have not yet studied the class of algorithms where message termination information diffuses throughout the system, as opposed to within the group; such a facility would allow us to *cancel* external timestamps that were no longer needed, and would further reduce the number of timestamps in the system, perhaps at the cost of extra messages.

These algorithms would benefit greatly from a performance study in a real, running system; without such a study, the real effects of group overlap patterns can only be guessed at.

7.3 Conclusion

We have presented a new scheme, the *bypass* protocol, for efficiently implementing a reliable, causally ordered multicast primitive. Intended for use in the ISIS toolkit, it offers a way to bypass the most costly aspects of ISIS while benefiting from *virtual synchrony.* The bypass protocol is inexpensive, yields high performance, and scales well. Measured speedups of more than an order of magnitude were obtained when the protocol was implemented within ISIS. Our conclusion is that systems such as ISIS can achieve performance levels competitive with the best existing multicast facilities — a finding contradicting the widespread concern that fault-tolerance may be unacceptably costly.

Bibliography

[AHL89] M.Stella Atkins, Garnik Haftevani, and Wo Shun Luk. An efficient kernel-level dependable multicast protocol for distributed systems. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 94–101. IEEE, 1989.

[BC90] Kenneth Birman and Robert Cooper. The isis project: Real experience with a fault tolerant programming system. Technical Report TR90-1138, Cornell University Computer Science Department, Ithaca, NY, July 1990.

[BCG91] Kenneth A. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: Group and multicast semantics. Technical report, Cornell University Computer Science Department, February 1991. Technical Report TR91-1185.

[BHG87] P.A. Bernstein, V. Hadzilicos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BJ87a] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, Texas, November 1987. ACM SIGOPS.

[BJ87b] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BJ89] Ken Birman and Tommy Joseph. Exploiting replication in distributed systems. In Sape Mullender, editor, *Distributed Systems*, pages 319–368, New York, 1989. ACM Press, Addison-Wesley.

[BSS90] Ken Birman, Andre Schiper, and Pat Stephenson. Fast causal multicast. Technical Report TR90-1105, Cornell University Computer Science Department, Ithaca, NY, April 1990. Submitted to ACM Transactions on Computer Systems.

[CASD86] Flaviu Cristian, Houtan Aghili, H. Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. Technical Report RJ5244, IBM Research Laboratory, San Jose, California, July 1986. An earlier version appeared in the 1985 Proceedings of the International Symposium on Fault-Tolerant Computing.

[CB89] Robert Cooper and Ken Birman. Supporting large scale applications on networks of workstations. In *Proceedings of 2nd Workshop on Workstation Operating Systems*, pages 25–28, Washington D.C., September 1989. IEEE, IEEE Computer Society Press. Order No. 2003.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coo90] Robert Cooper, 1990. Personal communication.

[Cri88] Flaviu Cristian. Probabilistic clock synchronization. Technical Report RJ6432, IBM Research Laboratory, October 1988.

[Cri90] F. Cristian. Reaching Agreement on Processor Group Memberhsip in Synchronous Distributed Systems. Technical Report RJ 5964, IBM Almaden Research Center, August 1990. Revised from March, 1988.

[CZ85] David Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[Dan89] Peter Danzig. Finite buffers and fast multicast. In *Proceedings of the ACM Conference on Measurement and Modelling of Computer Systems*, Berkeley, California, 1989. ACM SIGMETRICS.

[Dee88] S. Deering. Multicast routing in internetworks and extended LANs. In *Proceedings of theSymposium on Communications Architectures & Protocols*, pages 55–64, Stanford, California, August 1988. ACM SIGCOMM.

[Dij72] E.W. Dijkstra. Hierararchical ordering of sequential processes. In C.A.R. Hoare and R.H. Perrot, editors, *Operating Systems Techniques*. Academic Press, New York, 1972.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988.

[FM90] Alan O. Freier and Keith Marzullo. Mtp: An atomic multicast transport protocol. Technical Report TR 90-1141, Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, July 1990.

[GMS89] Hector Garcia-Molina and Annemarie Spauster. Message ordering in a multicast environment. In *Proceedings 9th International Conference on Distributed Computing Sytems*, pages 354–361. IEEE, June 1989.

[GT91] Ajei Gopal and Sam Toueg. Inconsistency and contamination with broadcasts and multicasts. part 1: Specifications. Technical Report TR 91-1189, Computer Science Department, Cornell University, February 1991.

[HHW89] A.A. Heddaya, M. Hsu, and W.E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1,2,3):35–57, Oct./Nov./Dec. 1989. Special issue on databases.

[HSSD84] Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, Vancouver, British Columbia, August 1984. ACM SIGOPS-SIGACT.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LLS90] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploting the semantics of distributed services. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 43–58, Qeubec City, Quebec, August 1990. ACM SIGOPS-SIGACT.

[Mar84] Keith Marzullo. *Maintaining the Time in a Distributed System.* Ph.D. dissertation, Stanford University, Department of Electrical Engineering, June 1984.

[Mat89] F. Mattern. Time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms.* North-Holland, 1989.

[MPS90] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol Based on Partial Order. Technical report, University of Arizona, 1990.

[OSS80] John Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: an experiment in distributed operating structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PGM85] F. Pitelli and Hector Garcia-Molina. Data processing with triple modular redundancy. Technical Report TR-002-85, Princeton University, June 1985.

[Pow83] Powell, Michael and Presotto, Dave L. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109, Bretton Woods, New Hampshire, October 1983. ACM. Proceedings published as *Operating Systems Review* 17(5).

[RB90] Aleta Ricciardi and Ken Birman. A formalism for fault-tolerant applications in asynchronous systems. In *Fourth SIGOPS European Workshop*, September 1990.

[RB91] Aleta Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. Technical Report Technical Report TR91-1188, Cornell University Computer Science Department, February 1991.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Sch88] Frank Schmuck. *The use of Efficient Broadcast Primitives in Asynchronous Distributed Systems.* Ph.D. dissertation, Cornell University, 1988.

[SES89] A. Schiper, J. Eggli, and A Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes on Computer Science 392*, pages 219–232. Springer-Verlag, 1989.

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[VRB89] Paulo Veríssimo, Luís Rodrigues, and Mário Baptista. Amp: A highly parallel atomic multicast protocol. In *Proceedings of theSymposium on*

Communications Architectures & Protocols, pages 83–93, Austin, Texas, September 1989. ACM SIGCOMM.

[WPE+83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70, Bretton Woods, New Hampshire, October 1983. ACM SIGOPS.