Reliable Broadcast Protocols*

Thomas A. Joseph
Kenneth P. Birman

TR 88-918
June 1988

TECHNICAL REPORT

Department of Computer Science
Cornell University
Ithaca, New York

Reliable Broadcast Protocols*

Thomas A. Joseph
Kenneth P. Birman

TR 88-918
June 1988

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

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037. Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) TR 88-918	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Kenneth P. Birman, Assist. Prof. CS Dept., Cornell University 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Defense Advanced Research Porject Agency/ISTO
6c ADDRESS (City, State, and ZIP Code)	7b ADDRESS (City, State, and ZIP Code) Defense Advanced Research, Project Agency Attn: TIO/Admin., 1400 Wilson Blvd. Arlington, VA 22209-2308
8a NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/ISTO 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code) See 7b.	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Reliable Broadcast Protocols

12 PERSONAL AUTHOR(S)
Thomas A. Joseph & Kenneth P. Birman

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical (Special	FROM ______ TO ______	June 1988	26

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This is a preprint of material that will appear in the collected lecture notes from Arctic 88, An Advanced Course on Operating Systems, Trömso, Norway, July 5-14, 1988. The lecture notes will appear in book form later this year.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Reliable Broadcast Protocols*

Thomas A. Joseph
Kenneth P. Birman

Department of Computer Science
Cornell University
Ithaca, NY 14853

June 1, 1988

This is a preprint of material that will appear in the collected lecture notes from *Arctic 88, An Advanced Course on Operating Systems*, Trömso, Norway, July 5-14, 1988. The lecture notes will appear in book form later this year.

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Arctic 88 Course Outline

1. Introduction
   1.1. Evolution of Distributed Systems
   1.2. Five perspectives on Distributed Systems
   1.3. Architecture
   1.4. Distribution transparency
   1.5. An architecture for selecting transparency
   1.6. Modelling systems

2. Technology
   2.1. Hardware Substrates and relevant software
   2.2. Networks

3. Communication
   3.1. Interprocess Communication
   3.2. Reliable Broadcast protocols
   3.3. Remote Procedure Call

4. Naming and Protection
   4.1. Naming
   4.2. Cryptography
   4.3. Protection

5. Concurrency and Consistency,
   5.1. Transactions
   5.2. Using transactions in distributed applications
   5.3. Theory of (nested) transactions
   5.4. Using replication to enhance availability and fault-tolerance in distributed systems
   5.5. Virtual synchrony for building distributed systems

6. File Systems
   6.1. Basics
   6.2. Why distributed file systems?
   6.3. Brief history
   6.4. Clarification of nomenclature
   6.5. Andrew File System
   6.6. Context
   6.7. Design
   6.8. Future

7. System Management
   7.1. Management Architectures
   7.2. Debugging, Reconfiguration control
   7.3. Accounting
   7.4. Effects of scale

8. Formal methods
   8.1. The tools of formal logic
   8.2. Representing behaviour in LOTOS
   8.3. High-level specifications for distributed programs

9. Conclusions

Chapter 1

Reliable Broadcast Protocols

The distinguishing feature of a distributed program is not just that its various parts are distributed over a number of processors but that these parts communicate with one another. The hardware in a distributed system allows a processor to send messages to other processors; the operating system usually extends this facility to allow a process on one machine to send messages to a process on another. The operating system may also provide facilities to set up virtual circuits between processes and may include protocols that ensure a certain degree of reliability in the communication. From the point of view of a programming language, however, these facilities are still rather low-level, and this has led to a search for appropriate high-level abstractions for inter-process communication. Some researchers suggest that distribution be completely hidden from the programmer. They argue for an abstraction that looks like a global shared memory. This abstraction has the advantage that it is simple to program with; writing a distributed program is no different from writing a non-distributed one. However, hiding distribution is not appropriate for all applications; some applications need to have explicit knowledge of location, either to obtain fault-tolerance or for better performance. Moreover, implementing the abstraction of a global shared memory on a network of computers could be extremely inefficient, especially if the network is large. It becomes increasingly difficult to justify the overhead of a shared memory abstraction as the network size becomes larger and a typical application runs only on a small fraction of the sites in the network.

A common high-level abstraction for inter-process communication is the *remote procedure call* (RPC), introduced by Birrel and Nelson [Birrel84]. A process communicates with another using an interface that looks just like a call to a procedure. The advantage of this abstraction is that it simplifies distributed programming by making communication with a remote process look like communication within a process. Its limitation, however, is that it is limited to two-way communication, namely communication between a calling process and a called process. Remote procedure calls are therefore most useful in distributed programs that fit the "client-server" model — client processes request services from server processes; server processes accept such requests and respond to each of them individually. However, RPC is not the most convenient abstraction when a distributed program is composed of a number of processes that have a high degree of inter-dependence on one another and the communication among them reflects this inter-dependence. In such programs the communication often takes place from one process to *a number of* processes rather than from a calling process to a called process, as in RPC's. An example of such a program would be a server that, for reasons of fault-tolerance or load sharing, is implemented as a group of processes on a number of sites. It would be convenient if a client requesting a service from such a server could send request to the group as a whole rather than have to be aware of the group's membership and pick one to communicate with. This is especially convenient if the server group could change its membership or location from time to time. Also, if the members of the group wish to divide up the work of responding to a request, each of them must ensure that its actions are consistent with what the other members are doing, and so they will need to communicate with one another. What is needed here is a facility that enables a process to send a message to *a set of processes*. We will call the act of sending a message to a set of processes a **broadcast**[1].

In its simplest form, a broadcast causes a copy of a message to be sent to each of the destination processes. What makes broadcasts interesting is that they must handle the possibility that some of the processes taking part in the broadcast may fail in the middle of the broadcast. For example, a failure could cause a broadcast message to be delivered to some but not all of its intended destinations — a possibility that never occurs when only two processes communicate with each other. To be useful to a program-

[1] Our use of the term *broadcast* does not refer to any hardware broadcast facility. On the contrary, we assume only that the network provides point-to-point communication. If the network does have a broadcast capability, some of the protocols described in this chapter can be optimized to take advantage of it.

mer, a broadcast must have well-defined behavior even when failures may occur. Broadcasts that provide such guarantees are called "reliable broadcasts." Reliable broadcasts are implemented using special protocols that detect failures and/or take compensating actions. Our definition of broadcast is general enough to cover protocols like 2- and 3-phase transaction commit protocols, and indeed some of the broadcast protocols we describe in this chapter are similar to these protocols. We begin our discussion with a description of our system model and our model of failures.

1.1 System model

Figure 1.1 shows our model of a distributed system. It consists of a number of processors (sites) connected to one another by a communications network. Each processor may have a number of user processes executing on it. There is no shared memory between sites and so the only form of communication between sites is through the network, which enables messages to be transmitted from any processor to any other processor in the system. Message transmission is asynchronous: sending and receiving processes do not have to wait for one another for communication to occur, and message transmission times are variable. Figure 1.2 shows the structure of the communication sub-system at each site[2] (the meaning of the arrows will be described later). The transport layer contains the hardware and the software that enables a message to be sent from one processor to another. We assume that the transport layer provides reliable, sequenced point-to-point communication. By this we mean that a message sent from one site to another is eventually delivered (unless the sending or the receiving site fails), and that messages between any pair of sites are delivered in the order they were sent. This form of reliability is achieved using protocols that sequence messages, detect lost or garbled messages (with high probability), and retransmit such messages. Many such protocols are described in [Tannenbaum81]. These protocols are not the subject of this chapter, which deals with the broadcast layer.

The broadcast layer implements the facility to send a message from one process to a set of processes, possibly on different machines. A process wishing to perform a broadcast presents the broadcast layer with a message and a list of destination processes for that message. The broadcast layer uses the

[2] The communication sub-system may be part of the operating system kernel, a separate system process, part of the user process, or any combination of these. We are concerned more with its *function* rather than its *location*.

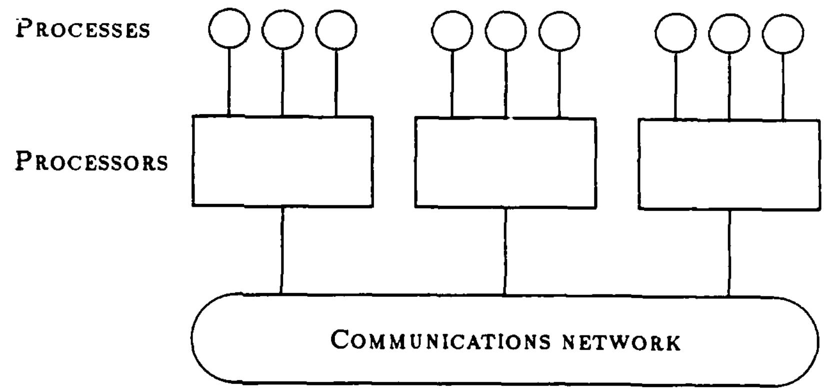

Figure 1.1: System model

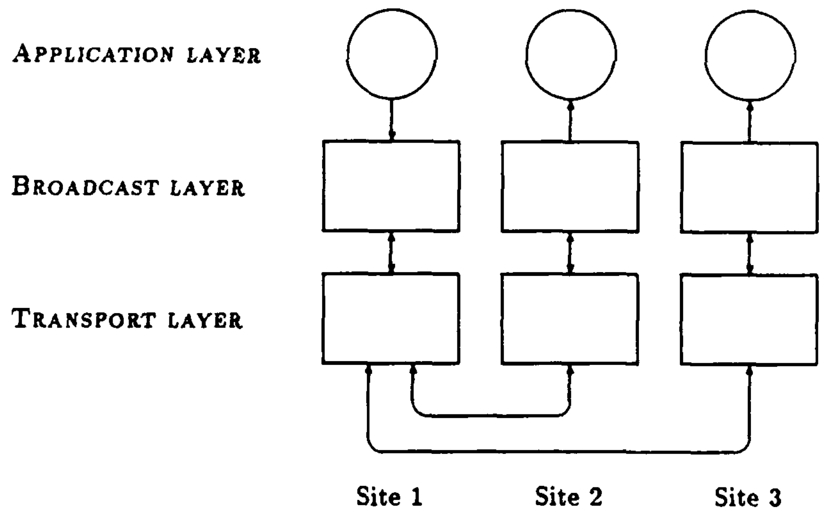

Figure 1.2: Communication sub-system

destination list to compute a set of sites that must receive this message, and uses the transport layer to send a copy of the broadcast message to each of these sites. It typically includes other information with the message, which is used by the broadcast layer at the receiving site. Depending broadcast protocol being executed, there may be further rounds of communication among the sites before the message is finally delivered to the destination processes at each of the sites. In what follows we will call the site from which a broadcast is made its **initiator**, and the sites to which it is sent its **recipients**. The arrows in Figure 1.2 shows a pattern of message exchange that could arise when a process at site 1 does a broadcast to processes at sites 2 and 3. In this figure, the broadcast layer at site 1 sends a message to the broadcast layers at sites 2 and 3, which engage in further communication with the broadcast layer at site 1 before they deliver the message to the application.

The protocol executed by the broadcast layer depends on the level of fault-tolerance it provides and on the way in which it orders the delivery of broadcasts relative to one another. We will consider a number of such broadcast protocols and examine their cost-performance trade-offs. We will begin with a protocol that achieves a simple form of fault tolerance and go on to more complex protocols that provide various ordering properties on broadcasts. Our detailed examples will be the broadcast protocols in the ISIS system [Birman87a,Birman87b], but we will also point to other similar protocols.

1.2 Failure model

To talk about reliable broadcasts we must first talk about what kinds of failures we are trying to overcome. The simplest failure model is the "crash model." In this model, the only kind of failure that can occur in the system is that a processor may suddenly halt, killing all the processes that may be executing there. Operational processes never perform incorrect actions, nor do they fail to perform actions that they are supposed to. Furthermore, all operational processes can detect the failure of a processor. For most of this chapter we assume that only crash failures can occur. There are a couple of reasons for restricting our attention to crash failures. First, the abstraction of crash failures can be implemented on top of a system subject to more complex failures by running an appropriate software protocol. The ISIS failure detector [Birman87a] and the protocol in [Schlichting83] are examples of such protocols. Second, techniques are available to automatically translate a protocol that tolerates crash failures into protocols that tolerate larger classes of failures [Neiger88]. Since protocols that tolerate only crash failures are simpler to develop and to understand, it is easiest to describe such protocols here and use these translation techniques to obtain more complex protocols if desired.

1.3 Atomic broadcast protocols

One of the simplest properties provided by a broadcast protocol is **atomicity**, that is, a broadcast message is either received by *all* destinations that do not fail or by *none* of them[3]. Moreover, non-delivery may occur only if the sender fails before the end of the protocol. An atomic broadcast protocol will never cause a message to remain undelivered at some non-faulty destinations if it has been delivered at some others (even if some destinations fail before the protocol completes). This is a very useful property because a process that receives such a broadcast can act with the knowledge that all the other intended destinations will also receive a copy of the same message. This reduces the danger of a recipient taking actions that are inconsistent with the others. Consider the case where a number of processes each maintain a copy of a replicated set of items. Let us say that a broadcast is made to these processes requesting them to add a particular item to this set. If an atomic broadcast protocol is used, each recipient can add the item to its copy of the set with the knowledge that all other destinations will also do so, and so their sets will all contain the same items. Without atomicity, the implementor of the replicated set will have to take steps to ensure that a failure will not cause some processes to miss updates, which would result in the copies of the set becoming inconsistent.

At first glance, an atomic broadcast protocol might seem trivial to implement, especially if the transport layer gives reliable point-to-point transmission. The initiator could simply send the message to each destination site, and a recipient could simply deliver it to any destination process at that site. But what happens if the initiator crashes after it has sent the message to some but not all of the destination sites? Now we are left in precisely the situation that we are trying to avoid: some destinations have received the message, while others have not. To make matters worse, the destinations

[3]Some researchers have used the term *atomicity* to refer to stronger properties. Here, we use it to only mean all-or-nothing delivery.

At the initiator:

send message m to all sites where there is a destination process

At a site receiving message m:

if message m has not been received already
  send a copy of m to all other sites where there is a destination process
  deliver m to any destination process at this site

Figure 1.3: A simple atomic broadcast protocol

that have not received the message have no idea that they should receive one. So it is necessary for one or more of the recipients to detect that the initiator has failed and to forward the message to the sites that did not receive it. This, of course, means keeping a copy of the message around for a while — at least until it is known that all destinations have received it. Since we cannot keep copies of messages around forever, some means must also be provided for a recipient to obtain the knowledge that a message has been received everywhere, so that it can then discard the message. So we see that what seemed to be a trivial problem turns out to be not so trivial after all!

Figure 1.3 gives a simple protocol that implements an atomic broadcast that tolerates crash failures. It is similar to the algoithm in [McCurley86]. When a site receives a message for the first time, it retransmits a copy of the message to all the destinations. Hence if a site receives a message and remains operational, all the destinations will receive a copy of the message. Thus atomicity is guaranteed. However, this property is achieved at the expense of increased communication because of the retransmissions. The protocol also takes up memory space because the message (or some part of it) must be stored at a recipient till all the retransmitted copies arrive, otherwise there will be no way of identifying these copies as duplicates of the first one. We could modify this protocol to retransmit messages only if the initiator is seen to fail. Then most of the extra communication would occur only when a failure occurs, which is more reasonable. But even when failures do not occur, this protocol would incur extra storage and communication costs. Each recipient must store the message until it is notified that it has been delivered at all its destinations, and this notification will require some message overhead. In general, depending on the properties that it achieves, a broadcast protocol will incur a cost in terms of latency (the time between when a message is sent and when it is delivered at its destinations), communication (because of extra messages or larger messages), and memory space.

1.4 More complex protocols

We considered a simple broadcast protocol that achieves atomicity. There are two directions in which one could go to arrive at more sophisticated protocols. One is to expand the class of failures that the protocol tolerates. The other is to consider protocols that provide stronger guarantees than atomicity. An example of a larger class of failures than crash failures is "omission failures." In this failure model, a faulty processor could occasionally fail to send or to receive messages that it should (or it could crash). This is a realistic model for processors connected by communications links that may lose messages, or if is is possible for their transmission buffers to overflow occasionally, causing messages to be lost. Interestingly enough, the protocol described above achieves atomicity even with this class of failures. We could go even further, and consider failure models like Byzantine failures, where processes may malfunction by sending out spurious or even contradictory messages. In the rest of this chapter, however, we restrict ourselves to crash failures, but consider protocols that are more complex because they achieve stronger properties than atomicity. For protocols that deal with omission and Byzantine failures, the reader is referred to [Perry86] and [Lamport82] respectively.

1.5 Ordered broadcast protocols

When we introduced atomicity, we considered the example of a number of processes cooperating to maintain a replicated set of items. We saw that atomicity was sufficient to ensure that all the copies of the set contained the same items. But what if the processes were maintaining a *queue* of items instead of a set? In this case, the *order* of the items is required to be the same in all the copies. Atomicity is not enough here because there are no guarantees on the order in which different broadcasts will be delivered (especially if they originate from different senders). Given a broadcast protocol that had the additional guarantee that messages will be delivered in the same order everywhere, implementing a replicated queue is simple: this protocol is used to broadcast items to the processes maintaining the queue, and each recipient adds items to its copy of the queue in the order that it receives them. Atomicity ensures that all operational copies will contain the same set of items; the ordering property ensures that these will be in the same order in all the copies. Without the ordering property, the implementor of a replicated queue will have to include code to ensure that all the copies agree on the order in which items are added to the queue, which makes developing this application a more difficult task. The availability of an ordered broadcast can simplify the implementation of many distributed applications, and consequently much work has been done in developing protocols for such broadcasts. We describe a few here.

If two sites broadcast messages to overlapping sets of destinations, it is possible for these messages to arrive at the overlapping destinations in different orders. The essential feature of an ordered broadcast protocol, then, is that an incoming message is delivered only when all the recipients have agreeed on how to order its delivery relative to other messages. This usually increases the latency, results in additional communication, and requires that the message be stored for the duration of the protocol. The algorithms we study below differ in the way they trade these costs off against one another.

The first protocol we study was proposed by Dale Skeen and is described in detail in [Birman87a] under the name *ABCAST*. It operates by assigning each broadcast a timestamp[4] and delivering messages in the order of timestamps. When a site receives a new message, it stores it in a pending queue, marking it as *undeliverable*. It then sends a message to the initiator with a *proposed timestamp* for the broadcast. This proposed timestamp is chosen to be larger than any other timestamp that this site has proposed or received in the past. (To make the timestamp unique, each site is assigned a unique number that it appends to its timestamps as a suffix). The initiator collects the timestamps from all the recipients, picks the largest of the values it receives, and sends this value back to the recipients. This becomes the *final timestamp* for the broadcast. When a recipient receives a final timestamp, it

[4] These timestamps need have no relation to real time; all that is required is an increasing sequence of numbers.

assigns the timestamp to the corresponding message in the pending queue, and marks the message as *deliverable*. The pending queue is then reordered to be in order of increasing timestamps. If the message at the head of the pending queue is deliverable, it is taken off the queue and delivered. This is repeated until the queue is empty or the message at the head of the queue is undeliverable (if there are deliverable message after this undeliverable one, they are *not* delivered at this time; they remain in queue until the messages ahead of them are all deliverable).

Figure 1.4 illustrates how this protocol works. Let us assume that (processes at) at three sites are trying to broadcast messages m_1, m_2 and m_3 to the same set of destinations at sites 1, 2 and 3. Assume that the largest timestamps seen at sites 1, 2 and 3 are 14, 15 and 16 resp. Step 1 shows the messages arriving at the recipients in different orders. They are all placed in the pending queues marked as undeliverable (u), with proposed timestamps as shown. Notice how the site number is used to disambiguate equal timestamps. In Step 2, the sender of m_1 collects its proposed timestamps (16.1, 17.2 and 17.3), computes the maximum (17.3), and sends this value to the recipients as the final timestamp. The recipients mark the message as deliverable (d) and reorder their pending queues as shown. Since there are no undeliverable messages ahead of m_1 at site 3, m_1 can be taken of the queue and delivered there, but it cannot be delivered at sites 1 and 2. Step 3 shows the pending queues after the sender of m_2 sends its final timestamp, and Step 4 shows the queues after the sender of m_3 does the same. At this point, all the messages can be taken off the pending queues and delivered. Observe that the messages are delivered at all sites in the order m_1, m_3 and then m_2, which was the order of their final timestamps.

The *ABCAST* protocol assigns each broadcast a unique final timestamp, and all messages are delivered in the order of their final timestamps. This ensures that broadcasts are delivered in the same order at all destinations. Because the sender picks the *largest* of the proposed timestamps, changing the timestamp of a message from its proposed one to the final one can only cause it to be moved *behind* other messages in a pending queue, and never ahead of them. So a message might have to wait for other messages to be delivered before it gets delivered, but there will never be a situation where it is necessary to deliver a message before one that has already been taken off the queue and delivered (which would cause this protocol to fail).

Let us examine the costs associated with this protocol. First, observe that a message cannot be delivered as soon as it is received; it has to remain in the pending queue until at least a second round of message exchange has

SITE 1 | SITE 2 | SITE 3

Step 1

Site 1:

m_3	m_1	m_2	
15.1	16.1	17.1	⋯
u	*u*	*u*	

Site 2:

m_2	m_1	m_3	
16.2	17.2	18.2	⋯
u	*u*	*u*	

Site 3:

m_1	m_3	m_2	
17.3	18.3	19.3	⋯
u	*u*	*u*	

Step 2

Site 1:

m_3	m_2	m_1	
15.1	17.1	17.3	⋯
u	*u*	*d*	

Site 2:

m_2	m_1	m_3	
16.2	17.3	18.2	⋯
u	*d*	*u*	

Site 3:

m_1	m_3	m_2	
17.3	18.3	19.3	⋯
d	*u*	*u*	

Step 3

Site 1:

m_3	m_1	m_2	
15.1	17.3	19.3	⋯
u	*d*	*d*	

Site 2:

m_1	m_3	m_2	
17.3	18.2	19.3	⋯
d	*u*	*d*	

Site 3:

m_3	m_2		
18.3	19.3	⋯	
u	*d*		

Step 4

Site 1:

m_1	m_3	m_2	
17.3	18.3	19.3	⋯
d	*d*	*d*	

Site 2:

m_3	m_2		
18.3	19.3	⋯	
d	*d*		

Site 3:

m_3	m_2		
18.3	19.3	⋯	
d	*d*		

Figure 1.4: The *ABCAST* protocol

occurred, and it has been assigned a committed timestamp. It has also to wait for all messages with smaller timestamps to be delivered. This represents the latency cost. Second, each broadcast results in a higher communication overhead beyond the act of sending the message to each destination site. Each recipient must also send proposed timestamps back to the initiator and the initiator must respond to all of them with the final timestamp. Finally, the message must be saved in the pending queue from the time it is received till the time it is delivered. This represents the storage cost. (Actually, the storage cost is higher than this. Some information about a message has to be maintained at each recipient until it is known that it has been delivered at *all* the destinations.)

We have not described how this protocol deals with failures. If a recipient crashes in the middle of the protocol, the initiator simply ignores it and continues the protocol without it. If the initiator fails, then one of the recipients must take over and run the protocol to completion.

Chang and Maxemchuck describe another family of protocols that achieve ordered reliable broadcasts [Chang84]. Their protocols do not require that the transport layer provide reliable point-to-point transmission — unreliable datagrams suffice because the retransmission of lost messages is built into their protocols. In these protocols, one member of each group of processes is assigned a token and is called the "token site." The token site assigns a timestamp for each broadcast, and broadcasts are delivered at all destinations in the order of their timestamps. This ensures that all broadcasts to a group are delivered in the same order at all members of the group. The protocols require that the token be periodically transferred from site to site. The list of possible token sites (called the "token list") is maintained at each of the token sites, and a token site passes the token to the next site in this list. The protocols operate correctly as long as the number of failures that occur is less than the size of the token list. The sites go through a "reformation phase" whenever the token list has to be changed — either because of a failure or because a new site is to be added to the list. The different members in this family of protocols have different values for the size of the token list and different rules for when the token is passed to the next site in the token list. These rules also determine the various costs for the protocols.

In the Chang and Maxemchuck protocols, a message may be committed and delivered to the destinations only when the token has been passed twice around the sites in the token list — at the end of the first round it is known that the message has been received everywhere, and at the end of the second it is known that it is committed everywhere. Thus the rate at which the token is passed from site to site (and the size of the token list) determines the latency cost as well as storage cost (as information about a message has to be stored until it is committed). If the token is passed rapidly, the latency and storage costs are minimized, but the communication costs go up. The communication costs may be reduced by passing the token infrequently, but this increases the latency and storage costs. In the limit, if the token is never passed, the additional communication goes down to one acknowledgement message per broadcast, but the latency and storage costs go up to infinity and fault-tolerance is lost.

1.6 Weaker orderings

Protocols that place a total order on all broadcasts are useful for many applications, but we have seen that they entail substantial latency, communication and storage costs. The natural question that arises is whether there are less expensive protocols that achieve something less that a total order on broadcasts and that are nevertheless useful for some applications. Within the ISIS system, much work was done to develop protocols that provided sufficient order to obtain consistency in replicated data, but which are asynchronous in the sense that messages can be delivered as soon as they arrive at a destination (without waiting for further rounds of communication). The advantage of using such a protocol to transmit updates to replicated data is that if there is a copy of the data at the sender site, then the latency to update this copy is almost zero (as a message can be sent from one site to itself with very little overhead). As a result, a local copy of replicated data can be updated at almost the same rate as a piece of non-replicated data (with some background overhead because of messages being sent to the sites with the other copies). We begin with an example.

Figure 1.5 shows processes P and Q sending broadcasts b_1 and b_2 to a group consisting of A and B. (The dashed lines represent the passage of time; the solid lines represent messages being sent.) For some applications, it may not be important that broadcasts from different processes be delivered in the same order, and it may be quite acceptable that A receives b_1 before b_2, while B receives b_2 before b_1, for example. On the other hand, because b_3 and b_4 were sent by the *same* process P and b_4 was sent after b_3, the broadcast b_4 could contain information that depends on b_3. For example, if A and B were maintaining a distributed data structure and b_3 were a message to initialize this structure and b_4 were a message that causes this data structure to be

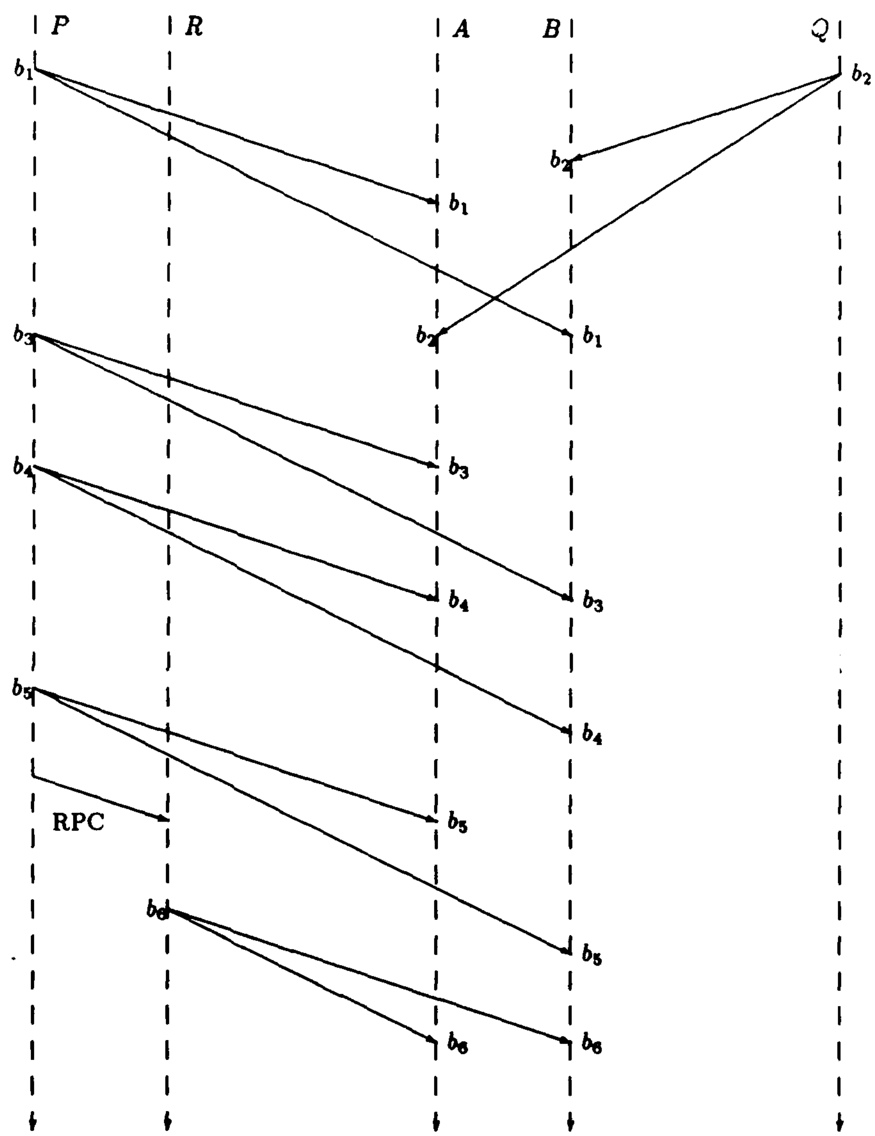

Figure 1.5: Unordered, FIFO, and causal broadcasts

accessed, then b_4 depends on b_3. Because of this causal dependency, it is desirable that b_4 be delivered after b_3 everywhere. The property we desire here is a FIFO property, namely that all broadcasts by the same process are delivered everywhere in the order that they were sent. This property is achieved automatically if the transport layer gives sequenced point-to-point communication. But what if P does a broadcast b_5, then does a remote procedure call to R, which then does a broadcast b_6? The broadcast b_6 is logically part of the same computation as b_5 and could have exactly the same causal dependency on b_5 as b_4 has on b_3 (b_5 could be a message to initialize a data structure and b_6 one to accesses it). Unfortunately, because b_5 and b_6 originate in different processes, the FIFO property makes no guarantee on the order in which they will be delivered. This is especially unfortunate because if b_6 were a broadcast from within a *local* procedure call, a programmer developing this application could take advantage of the fact that the deliveries would be ordered, but just because the procedure call happened to be remote, his task becomes far more complicated. What would be useful here is a broadcast protocol that guarantees that if the initiation of a broadcast b is causally dependent (as described above) on the initiation of a broadcast b', then b will be delivered after b' everywhere. We first need to formalize the notion of causal dependency.

An event a occurring in a process P can affect an event b in a process Q only if information about a reaches Q by the time b occurs there. In the absence of shared memory, the only way that such information can be carried from process to process is through messages that travel between them. Accordingly, as in [Lamport78], we can define the potential causality relation $a \rightarrow b$ (b is potentially causally dependent on a) to be the transitive closure of the two relations $\xrightarrow{1}$ and $\xrightarrow{2}$ defined below:

1. $a \xrightarrow{1} b$ if a and b are events that occur in the same process and a occurs before b.

2. $a \xrightarrow{2} b$ if a is the sending of a message and b is the receipt of the same message.

Informally, if a is an event in process P and b is an event in process Q, then $a \rightarrow b$ if and only if there is a sequence of messages $m_1, m_2, \ldots, m_n$ and processes $P = P_0, P_1, P_2, \ldots, P_n = Q$ ($n \geq 0$) such that message m_i travels from P_{i-1} to P_i and is delivered to P_i before m_{i+1} is sent from there. Also, m_1 is sent from P after event a occurs there, and m_n is delivered to Q before b occurs there. It is the existence of this sequence of messages that enables information about a to be carried to Q and so makes b potentially causally dependent on a.

What is needed, then, is a broadcast protocol that ensures that if $send(b_1) \rightarrow send(b_2)$, then b_2 will be delivered after b_1 at all overlapping destinations. The protocol *CBCAST* (for causal broadcast) described in [Birman87a] achieves this. The protocol in [Peterson87] is similar. The easiest way to explain the *CBCAST* protocol is to start with a grossly inefficient version and derive the actual protocol from it. Imagine that for each process P the broadcast layer at its site keeps a buffer containing every message P has ever sent or received (in order). Any time a broadcast b is initiated by P, this buffer will then contain every message that could have causally affected b. Whenever any message m is sent from a site, the protocol sends the entire contents of these buffers along with m (i.e. it piggybacks the buffers onto m). At the receiving site, the broadcast layer adds the piggybacked messages to all its buffers (preserving their order, but discarding duplicates) even if the piggybacked messages are not destined for any process at that site. It then delivers (in order) any messages destined for processes at that site, the last of which will be m.

The reason why the protocol described above works is simple. If b_1 is initiated by process P at site S and b_2 by Q at T and if $send(b_1) \rightarrow send(b_2)$, then there must be a sequence of messages as described above from S to T. The protocol ensures that b_1 will be piggybacked on this sequence of messages (and possibly on other messages as well) and so b_1 will reach T and before b_2 is sent. Since b_1 will be in Q's buffer when b_2 is sent from there, b_1 will be piggybacked on b_2 and will hence be delivered before b_2 at any overlapping destination.

The problem with the scheme described above, of course, is that the amount of information to be piggybacked grows indefinitely. There are a number of ways in which the protocol described above can be optimized. First, the buffers can be maintained on a per-site basis instead of a per-process basis. This reduces the storage overhead. Second, a message does not have to be piggybacked to a site if it has been sent there already. More importantly, messages do not have to be piggybacked once it is known that they have reached all their destinations, because they will be discarded on arrival anyway. This means that a message needs to be piggybacked only from the time a broadcast is initiated till the time it reaches at all the destination sites. If we call this time period δ, piggybacking need occur only if broadcasts are being made at a rate of more than one every δ time units. δ is usually a very small window and so unless broadcasts are being made rapidly one after another, there need be very little actual piggybacking. The initiator can stop piggybacking a message when its transport layer receives an acknowledgement from all the recipients; other sites must continue to do so until they are informed that the message has reached all its destinations. The performance of this protocol thus depends on how effectively this information is propagated to sites that have a copy of this message. This issue can be avoided by piggybacking a message only on messages going directly to the destination sites. Other sites are instead sent a small descriptor that identifies the message. If a destination receives a descriptor before it receives the actual message, it must wait for the message to arrive before delivering any message that may causally depend on it.

Messages sent using the *CBCAST* protocol can be delivered as soon as they reach a destination site. There is no need to wait for additional rounds of communication and hence no latency cost (except to the extent that transmitting larger messages may take a slightly longer time). The protocol requires no additional messages besides those required to get the message from the initiator to the destinations, but it does increases the message size. In most systems, the *number* of messages (and not their size) is the dominant factor in the communication cost[5] and so the communication overhead is minimal. The protocol does have a storage cost because the messages have to be buffered while piggybacking is going on.

FIFO broadcasts preserve the order of causality in a computation that runs at one site; causal broadcasts generalize this to distributed computations. Causal broadcasts can be used to order deliveries when all broadcasts to a group arise from a computation with a single thread of control, but this thread of control may span several sites (e.g. because of remote procedure calls). They can also be used when broadcasts to a group arise from different computations, but these computations have some other form of synchronization relative to one another. An example of this would be broadcasts to a group that arise from within nested transactions whose sub-transactions may run on different sites. Here the broadcasts arising from sub-transactions of any one transaction will be ordered because they are causally related; broadcasts arising from different transactions will be ordered because of the concurrency control mechanism used to implement nested transactions.

[5]This is true only up to a point. If a message size gets very large, it may have to be fragmented into a number of smaller packets before being transmitted.

1.7 Real time delivery guarantees

Another property that may be useful in a reliable broadcast protocol is that delivery will occur within a specified amount of time after the initiation of the protocol. This is especially useful in real time systems and in control applications, where a broadcast that arrives too late may not produce the desired response. If a broadcast is being made to a set of processes to instruct them to each begin some action, it might also be desirable that broadcast deliveries occur within a known time interval of one another, so that their actions take place with some degree of simultaneity. The protocols described earlier make no such guarantees — they ensure that broadcasts will be eventually delivered to all non-faulty destinations, but delivery could take arbitrarily long.

In [Cristian86], Cristian *et al* describe several broadcast protocols that provide real time delivery guarantees. For such protocols, one needs to have timing bounds on various aspects of system behavior, for example, a bound on the time it takes for the system to schedule a process for execution, a bound on the time it takes for a message to travel from one site to another, the ability to schedule an event to occur within a certain time, etc. Given such bounds, one can devise broadcast protocols by taking into account worst-case timing behavior. For example, one can achieve simultaneous delivery by timestamping each broadcast with the sending time t and computing Δ, the maximum time it can take for a message to reach a destination. Now if a broadcast is buffered at each destination and delivered only at time $t + \Delta$, simultaneous delivery is achieved. It should be noted that "simultaneous" here means that the processors will deliver a broadcast at the same time *as read off their own clocks*. In practice, the clocks of individual processors will differ somewhat from real time, and at broadcast will *not* be delivered everywhere at exactly the same instant. However, by using algorithms such as described in [Srikanth87], the clocks of the various processors can be synchronized to the degree required, thus achieving the desired level of simultaneity.

The calculation of the constant Δ must take into account possible differences in clock values as well as possible scheduling and message transmission delays, and is described in detail in [Cristian86]. In addition, this calculation must account for faulty system behavior. One kind of possible failure is a "timing fault." Recall that the protocols were based on timing bounds for certain system activities. If the system violates these timing bounds (e.g. a message takes longer to be delivered than the assumed upper bound). we have a timing fault. We could also consider other classes of failures like omission or Byzantine failures. Cristian *et al* describe protocols to achieve reliable real time broadcasts that tolerate increasingly higher classes of faults, from no faults at all to Byzantine faults.

There is a basic difference between these protocols and the ones described earlier. The earlier protocols use explicit message transfer to ensure that a broadcast has arrived at all its destinations and to agree on an order for its delivery. These protocols, on the other hand, use the passage of time (and knowledge of timing bounds on system behavior) to obtain the same information. As a result, the latter protocols will, in general, have a lower communication cost. However the latency and storage costs are based on worst-case system behavior. If the variance in the duration of system events (e.g. message transmission) is low and one has accurate estimates of these times, the latency and storage costs are likely to be low. On the other hand, if the variance is high (as would happen if the load on the system is variable), then the fact that these costs are based on worst-case behavior might make them unacceptably high. The latency is especially critical, because the perceived speed of an application performing broadcasts depends on this.

1.8 Broadcasts to dynamically changing groups

Until now, we have considered broadcasts made to a fixed set of destinations. The protocols described above assume that the set of destinations is known when a broadcast is initiated and that it does not change. For many applications, it is useful to be able to broadcast a message to a "process group" — a symbolic name for a set of processes whose membership may change with time. Such a group may implement some service like a document formatting service or a compile service. The reason for implementing such a service using a group of processes instead of a single one may be to divide up the work of responding to a user's request over a number of machines, to obtain faster response time by executing a user's request on a machine best suited to that particular request, to have the service remain available despite the failures of some machines, or any combination of these. New members may join the group as the number of requests on the service increases or as idle machines volunteer their cycles for the service. Members may leave the group as the load on the service decreases or when a machine crashes. It is useful if a user of such a service can use the process group name to communicate with the service without needing to know the membership of the group or where the members are located.

To implement broadcasts to process groups, the system must provide a facility for mapping process group names to sets of processes, and provide some semantics for what it means to perform a broadcast to a group whose membership might be changing as the broadcast is under way. The V system [Cheriton85] provides a means to broadcasts to process groups, but there are no ordering guarantees on broadcast message delivery. Also, if the membership changes as a broadcast is in progress, it is possible for the broadcast to be delivered to some intermediate set of destinations that is neither the old membership nor the new one. In [Cristian88], Cristian discusses the problem of agreeing on group membership in systems that have timing bounds on their behavior, and describes a solution based on the protocols described in [Cristian86]. The ISIS system provides an addressing mechanism that permits ordered broadcasts to be made to dynamically changing process groups. In addition to causal or totally ordered message delivery, ISIS guarantees that if the membership of a process group is changing as a broadcast is under way, the broadcast message will be delivered either to the members that were in the group before the change or to those that were in the group after the change, and never to some intermediate membership. In other words, it is never possible for a broadcast to a group to be delivered to some processes after they have seen a change in the group membership and to other processes before they have seen that change. Let us see why this property is useful.

Consider Figure 1.6, which shows processes executing in an environment where broadcast delivery is *not* ordered relative to group membership changes. We see a process P using a broadcast to present a task made up of 6 sub-tasks to a group currently consisting of processes A and B. The group divides up the task equally, with the first process[6] taking the first set of sub-tasks, and so on. Let us suppose that P sends the group another similar task around the same time that process C attempts to join the group. The figure shows A receiving the task before it knows that C joined the group, while B and C receive the task after they see C join. Consequently, A divides the task on the assumption that the group consists of two members, while B and C do so on the assumption that there are three members. The result is an inconsistent division of the task. In this case sub-task 3 gets executed

[6] Any deterministic ordering on process names may be used. Here we have used the lexicographic order.

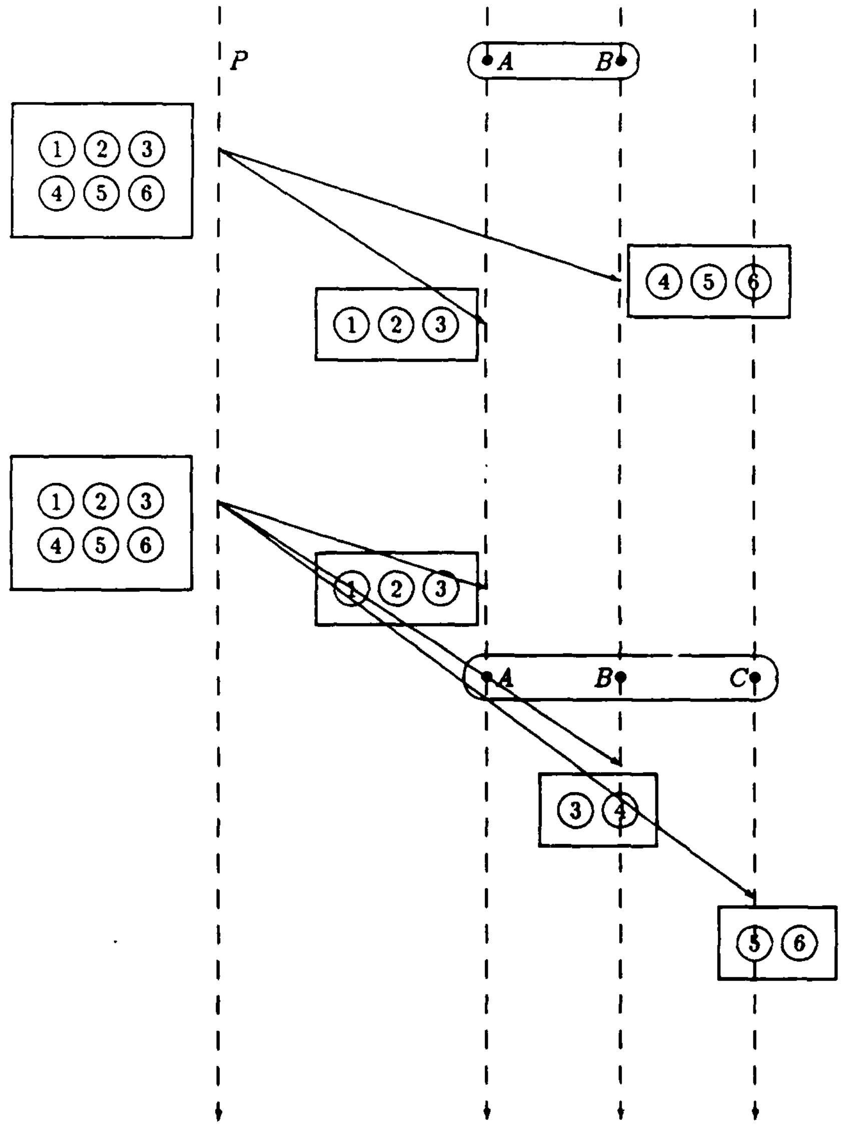

Figure 1.6: Unordered group membership changes

twice (which may or may not be acceptable), but if this anomaly arose as a member was *leaving* the group instead of joining, some sub-tasks might end up not being executed by any member (which is clearly unacceptable). The only way to avoid this problem is for the group members to execute some protocol that ensures that they all have the same view of the group membership before they respond to any request. However, if the broadcast delivery were ordered relative to group membership changes, this problem would not have arisen in the first place.

What the example illustrates is that if broadcast delivery is not ordered relative to group membership changes, and if the members of the group have to coordinate the actions they take in response to an incoming request, then additional protocols are needed to ensure that they respond based on consistent views of the group membership. This makes the task of the person programming such an application difficult and adds to the complexity of the algorithms used. On the other hand if broadcast delivery *is* ordered relative to group membership changes, there are no such problems. Each member can respond to an incoming request based on its view of the group membership, with the assurance that when the other members receive the same request, they will all have exactly the same view, and will hence take consistent actions. Note that group membership may change not only when a process voluntarily joins or leaves a group, but also when a process drops out of a group because of a *failure*. To be completely useful, the process group mechanism must order broadcast deliveries with respect to the latter kind of group membership change as well. This might seem impossible to achieve because the system has no control over when failures occur, but in fact it can be achieved because what is important is that each process *observes* group membership changes and broadcast deliveries in the same order, or that each process *detects* failures and broadcast deliveries in the same order, and not that the failure actually occurs in an orderly fashion.

To explain how the process group mechanism is implemented in the ISIS system, we will first describe a simplistic mechanism and then show how it may be modified. For now assume that every site in the system has a table containing the names of every existing process group and their current membership. When a process at a site initiates a broadcast to a group, the system simply obtains a list of the current members from the table at that site and executes the relevant broadcast protocol using that list. When a process joins or leaves a group, the tables must all be changed. This is done using a special broadcast protocol whose deliveries are ordered consistently relative to *all* other kinds of broadcasts. In ISIS, the other kinds of broadcast are *ABCAST* and *CBCAST*, and the corresponding special broadcast protocol is called *GBCAST* (for group broadcast). An interlocking mechanism is also required to ensure that broadcasts that have been initiated using the old membership list are delivered before a *GBCAST* is delivered. When a *GBCAST* is delivered at a site, the table at that site is changed and all interested processes are notified of the membership change. Since *GBCAST* is ordered relative to all other broadcasts, all processes observe membership changes in a way that is ordered consistently with respect to other broadcast deliveries. It is, of course, impractical to maintain group membership lists on a system-wide basis and do a system-wide broadcast whenever the membership of any group changes. What ISIS actually does is to maintain information about the membership of a group at the sites where members reside (member sites) and optionally at a few other sites (client sites). Membership changes are broadcast using *GBCAST* only to member and client sites. This ensures that membership changes are ordered relative to broadcasts that originate from member or client sites. If a broadcast is made to a group from a site that is neither a member nor a client site, the system first obtains the current membership list from elsewhere (or uses an old but possibly inaccurate cached list) and then executes the relevant broadcast protocol. This leaves open the possibility that the membership may have changed between when the broadcast message was initiated and when it is about to be delivered. The system detects this if it happens and does not deliver the message. Instead, it sends the new membership list to the initiator site, which then restarts the broadcast protocol with this new set of destinations. This protocol will continue to iterate until the membership list remains unchanged from the time the broadcast is initiated till the time it is delivered. This kind of iteration increases the possible latency cost. This cost can be reduced by increasing the number of client sites, but the trade-off is that membership changes now become more expensive.

1.9 Conclusion

In this chapter we considered a number of reliable broadcast protocols, with different kinds of ordering and delivery guarantees. Developing applications that are distributed over a number of sites and/or must tolerate the failures of some of them becomes a considerably simpler task when such protocols are available for communication. Indeed, without such protocols the kinds of distributed applications that can reasonably be built will have a very limited scope. As the trend towards distribution and decentralization continues, it will not be surprising if reliable broadcast protocols have the same role in distributed operating systems of the future that message passing mechanisms have in the operating systems of today. On the other hand, the problems of engineering such a system remain large. For example, deciding which protocol is the most appropriate to use in a certain situation or how to balance the latency-communication-storage costs is not an easy question. It is our hope that as our experience with broadcast based systems grows, we will begin to gain insight into some of these problems.

Bibliography

[Birman87a] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems 5*, 1 (Feb. 1987), 47–76.

[Birman87b] Kenneth P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Austin, TX, Nov. 1987, 123–138.

[Birrel84] A. D. Birrel and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39–59.

[Chang84] Jo-Mei Chang and N. F. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 251–273.

[Cheriton85] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems 3*, 2 (May 1985), 77–107.

[Cristian86] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *IBM Research Report*, RJ 5244 (54244), July 1986.

[Cristian88] Flaviu Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *IBM Research Report*, RJ 5964 (59426), March 1988.

[Lamport78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558-565.

[Lamport82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.

[McCurley86] R. McCurley and F. B. Schneider. Derivation of a Distributed Algorithm for Finding Paths in Directed Networks. *Science of Computer Programming 6*, 1 (Jan. 1986), 1–10.

[Neiger88] Gil Neiger and Sam Toueg. Automatically Increasing the Fault-Tolerance of Distributed Systems. *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, Toronto, Ontario, Aug. 1988.

[Perry86] Kenneth J. Perry and Sam Toueg. Distributed Agreement in the Presence of Processor and Communication Faults. *IEEE Transactions on Software Engineering, SE-12*, 3 (Mar. 1986), 477–482.

[Peterson87] Larry L. Peterson. Preserving Context Information in an IPC Abstraction. *Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems*, March 1987, Williamsburg, VA, 22-31.

[Schlichting83] Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems 1*, 3 (Aug. 1983), 222–238.

[Srikanth87] T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the ACM 34*, 3 (July 1987), 626–645.

[Tannenbaum81] Andrew S. Tannenbaum. Computer Networks. *Prentice-Hall, Inc.*, Englewood Cliffs, N. J. 07632.