The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems

Frank Bernhard Schmuck
Ph.D. Thesis

TR 88-928
August 1988

NAG2-593

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

THE USE OF EFFICIENT BROADCAST PROTOCOLS IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

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

Frank Bernhard Schmuck

August 1988

The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems

Frank Bernhard Schmuck, Ph.D.

Cornell University 1988

Reliable broadcast protocols are important tools in distributed and fault-tolerant programming. They are useful for sharing information and for maintaining replicated data in a distributed system. However, a wide range of such protocols has been proposed. These protocols differ in their fault tolerance and delivery ordering characteristics. There is a tradeoff between the cost of a broadcast protocol and how much ordering it provides. It is, therefore, desirable to employ protocols that support only a low degree of ordering whenever possible. This dissertation presents techniques for deciding how strongly ordered a protocol is necessary to solve a given application problem.

We show that there are two distinct classes of application problems: problems that can be solved with efficient, asynchronous protocols, and problems that require global ordering. We introduce the concept of a *linearization function* that maps partially ordered sets of events to totally ordered histories. We show how to construct an asynchronous implementation that solves a given problem if a linearization function for it can be found.

We prove that in general the question of whether a problem has an asynchronous

solution is undecidable. Hence there exists no general algorithm that would automatically construct a suitable linearization function for a given problem. Therefore, we consider an important subclass of problems that have certain commutativity properties. We present techniques for constructing asynchronous implementations for this class. These techniques are useful for constructing efficient asynchronous implementations for a broad range of practical problems.

Biographical Sketch

Frank Schmuck was born in Bonn and grew up in Hamburg, West Germany. He received his B.S. in Physics from the Christian Albrechts Universität zu Kiel, West Germany in 1983. He first came to the United States in the fall of 1983 as an exchange student at the Pennsylvania State University. During his stay there his interests shifted from Physics to Computer Science, and he decided to stay longer than originally planned to get a Masters degree, which he received in the spring 1985. After spending half a year working for Siemens in Munich, Germany, he returned to the United States to pursue a Ph.D. in Computer Science at Cornell University.

To my parents

Acknowledgements

First of all, I wish to express my gratitude to my advisor, Ken Birman, for his continuous support and encouragement. I would like to thank Sam Toueg and Fred Schneider for many helpful discussions and comments on this work. I also thank Dexter Kozen and Lars Wahlbin for serving on my committee. My thanks go to Patrick Stephenson and David Basin for reading a draft of this dissertation.

I am grateful to my wife, Sabine, who had to leave her friends and family in Germany to follow me to Ithaca. Finally, I wish to thank all my friends who made my stay in Ithaca enjoyable, especially David for introducing me to the art of juggling.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Broadcast protocols are useful tools for distributed and fault-tolerant programming. However, a wide range of such protocols has been proposed, differing in their fault tolerance and delivery ordering characteristics. This thesis describes techniques for chosing the type of broadcast that will maximize the performance of an application without compromising its correctness.

This work was motivated by the ISIS system, a toolkit for building fault-tolerant distributed applications. All tools provided by ISIS are based on a set of broadcast communication primitives. These primitives as well as the tools built from them are made available to the application programmer. One objective of this thesis is to gain an understanding of the theoretical foundations of the ISIS system, and thereby help the programmer in selecting and using the tools provided by ISIS. The interested reader is referred to [BJ87a,BJS88] for a description of ISIS.

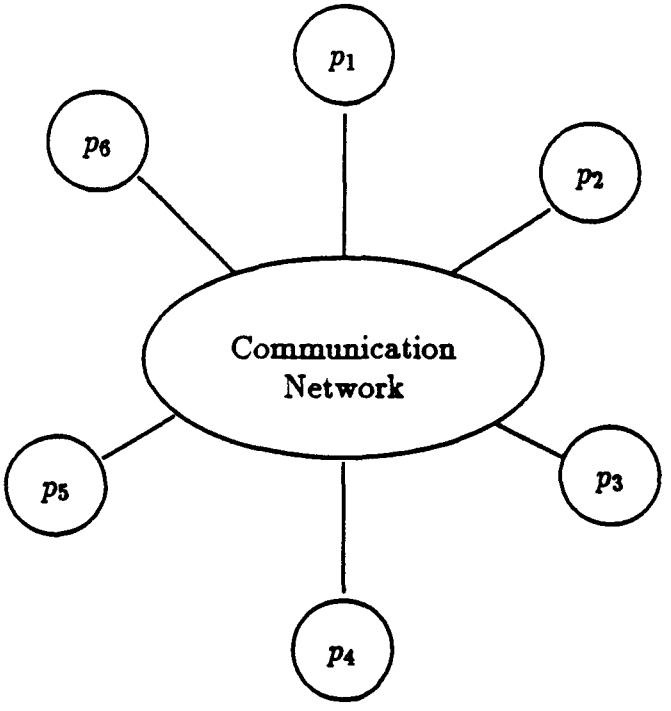


Figure 1.1: Distributed system

1.1 Distributed Systems

A distributed system consists of a set of independent processors, $p_1, \ldots, p_n$, connected by a communication network (see Figure 1.1). In such a system, processors exchange information only by sending messages. There are several parameters that determine the characteristics of the system:

- **Network topology**: Some pairs of processors can communicate directly. Messages between other pairs of processors have to be routed through one or more intermediate processors. A network in which every pair of processors can communicate directly is called *completely connected.*

- **Message ordering:** Some communication protocols do not make any guarantees about the order in which messages are delivered. Other protocols provide FIFO ordering, i.e., messages are delivered in the order they were sent.

- **Message reliability:** Message channels may be *reliable* (all messages sent are delivered correctly), subject to *omission failures* (messages may be lost), or subject to *Byzantine failures* (messages may be lost or corrupted). Furthermore, there may or may not be an upper bound on the time between the sending of a message and its delivery.

- **Processor reliability:** A processor may fail in several ways. It may stop without taking incorrect actions (*fail-stop*) [SS83], fail to send or receive some messages (*omission fault*) [Had84], or behave arbitrarily (*Byzantine fault*) [LSP82, SD83].

In this dissertation we assume a completely connected network with reliable message delivery. This decision is justifiable on practical grounds: data-link protocols and network routing protocols satisfying these assumptions are well understood [Tan81]. In general, we do not assume an upper bound on message delays, nor do we assume that processors have synchronized clocks. A system with these characteristics (unbounded message delays, no synchronized clocks) is called *asynchronous*.

Processors may experience failures, but we restrict ourselves to non-byzantine failure modes. Processor omission faults can be treated in the same way as the loss of messages in the communication network. Therefore, we consider only crash failures (fail-stop processors).

1.2 Objectives

Many applications running in a distributed system require processors to share information. Often it is also desirable to replicate information at different sites to avoid data loss should a failure occur. Useful tools for sharing information and for maintaining replicated data are *reliable broadcast protocols*. Such protocols propagate information from one processor to a set of destination processors in such a way that all operational destinations receive this information despite failures in the system. This property is called *reliable message delivery*. In addition to this, a broadcast protocol may also provide a form of *message ordering*. The strongest form is *atomic ordering*. An atomic broadcast protocol guarantees that all messages are received in the same order everywhere. An example of a weaker form of ordering is FIFO. A FIFO broadcast guarantees that two messages sent by the same processor are received everywhere in the order they were sent. Messages sent by different processors, however, may be received in different orders at different sites.

There is a tradeoff between how much ordering a protocol provides and how much synchronization delay is necessary to implement this ordering. A FIFO broadcast, for example, can be implemented efficiently on top of unordered message channels by adding a *sequence number* to every message. An atomic broadcast, on the other hand, is much more costly to implement in the systems we study. It requires two or more phases of message exchanges between processors before a message can be delivered. It is, therefore, desirable to employ protocols that support only a low degree of ordering whenever possible. This dissertation presents techniques for deciding how strongly ordered a protocol has to be in order to solve a given application problem.

1.3 Outline

This dissertation consists of seven chapters. Chapter 2 describes several different forms of broadcast protocols known in the literature and discusses their benefits and costs.

Chapter 3 introduces a formalism for specifying an application problem in a distributed system, and presents a model for broadcast-based implementations that solve such problems.

Chapter 4 investigates conditions under which a specification has an asynchronous implementation. It is shown that if such an implementation exists, it can be expressed in a canonical form.

Chapter 5 proves that in general the existence of an an asynchronous implementation for a given problem is undecidable. However, we identify a subclass of specifications that captures a broad range of practical problems. The defining characteristic of specifications in this class is that they have certain commutativity properties. We describe methods for finding asynchronous implementations for specifications in this class.

Chapter 6 examines how processor failures can be integrated into our model and shows how this affects the results of Chapter 4 and Chapter 5.

Chapter 7 summarizes our results and discusses future extensions of our work.

Throughout the thesis we use an example that is first introduced in Chapter 3. Appendix A contains a comprehensive presentation of this example in which we collect the different elements addressed throughout the thesis into one discussion.

Chapter 2

Reliable Broadcast Protocols

Because this dissertation is about selecting among different forms of broadcast protocols, we devote this chapter to giving an overview of several variants of broadcast protocol.[1] Such protocols have two distinct properties:

- *Reliability:* The protocol ensures that a message that is broadcast will eventually reach all its destinations, even if failures occur while the protocol is running.
- *Ordering:* Some protocols make guarantees about the order in which different broadcast messages are received at different destination sites.

We will describe how these features can be implemented on top of a network that provides only point-to-point communication between processors. In our discussion we will first concentrate on the ordering aspect. The following section will present different ordering properties and describe how such properties can be implemented

[1] The term "broadcast" is often used to mean that a message is sent to *all* processors in the system. We will use it in the more general sense of sending a message to some subset of all processors. This is often called a *multicast*.

in a completely reliable system in which processors do not fail. In Section 2.2 we will examine how the different types of protocols can be made fault-tolerant.

2.1 Broadcast Ordering

2.1.1 Unordered Broadcast

The simplest way of broadcasting a message is to just send a copy of that message to every destination processor individually. This form of broadcast does not provide any form of ordering. Figure 2.1 illustrates this. It shows a system with four processors. Time proceeds from left to right, and the diagonal lines represent messages. The figure shows p_1 broadcasting two messages (a and b) to p_2, p_3, and p_4. The two messages arrive in the same order at p_2 and p_3, but p_4 receives them in a different order. Because this form of broadcast does not guarantee any specific order of delivery, we call it an *unordered broadcast*, or simply BCAST.

2.1.2 Fifo Broadcast

If the underlying communication network provides FIFO message channels, then the protocol just described will satisfy a stronger ordering property: All messages broadcast by the same processor will be delivered in the same order everywhere, namely the order they are sent. Even if the network does not provide FIFO message channels, it is not difficult to implement FIFO ordering by adding a sequence number to every message [Tan81]. We call this a FIFO broadcast, or FBCAST for short. In Figure 2.2, for example, processor p_1 broadcasts two messages, first a then b. Processors p_3 and p_4 receive these messages in the order they were sent. Broadcasts B_2 and B_3, however, are sent by different processors; such broadcasts may be delivered in different orders at different sites, as shown in the example.

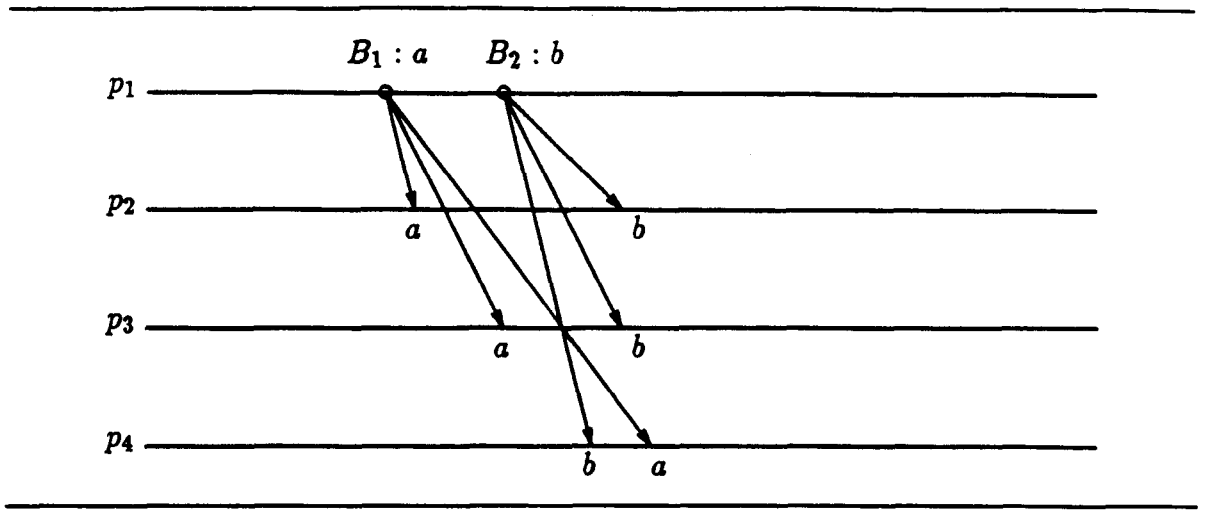


Figure 2.1: Unordered broadcast

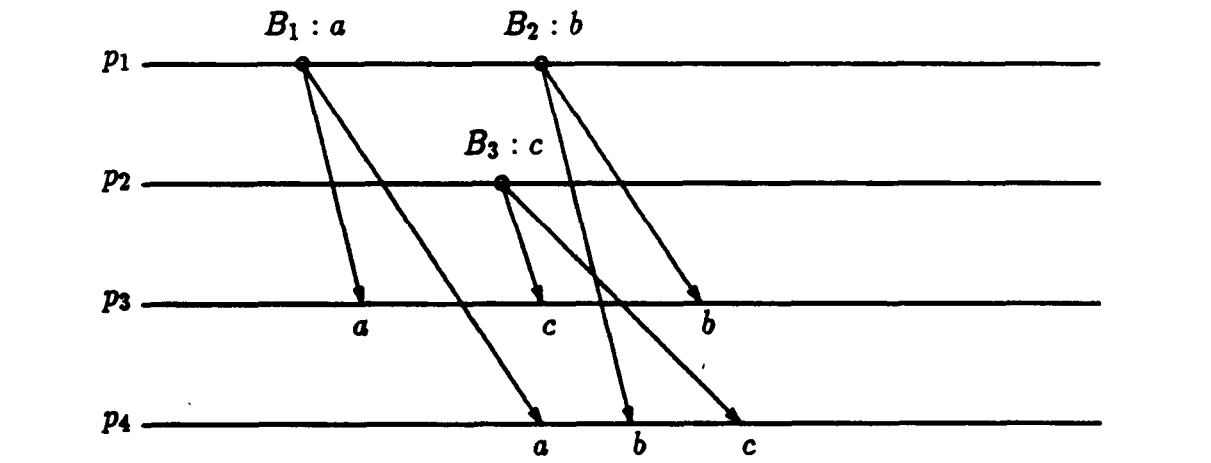


Figure 2.2: FIFO broadcast

2.1.3 Atomic Broadcast

...casts are often used for updating information that is replicated at several sites. FIFO ordering may not be enough if different processors broadcast update messages independently. In this situation, two update messages could be delivered in different orders at different sites, leading to inconsistencies. This can be avoided by using a stronger protocol that would guarantee that all messages are delivered in the same order everywhere, even if they were sent independently by different processors. Such a protocol is called an *atomic* broadcast protocol, or ABCAST for short. Figure 2.3 illustrates the behavior of an ABCAST. It shows two messages broadcast independently by p_1 and p_2. Both messages are received in the same order at p_3 and p_4 (first b, then a in this example).

There are several well known techniques for implementing ABCAST in asynchronous systems. Figure 2.4 illustrates a protocol due to Chang and Maxemchuk [CM84] in which every message is broadcast in two phases. A processor wishing to broadcast a message sends this message to one distinguished processor, say p_1 (first phase). p_1 then forwards the message to its destinations by means of a FBCAST (second phase). This way all broadcast messages are delivered in the order they were received and forwarded by p_1. A different, more symmetric atomic broadcast protocol due to Skeen is described in [BJ87b]. This method uses a three-phase protocol as illustrated in Figure 2.5. Every processor maintains a message delivery queue; when a broadcast message is received (phase one of the protocol), it is added to the queue, but not yet delivered to the application program running at that site. The recipient assigns a temporary "priority number" to the message and returns this number to the sender of the broadcast (phase two). The recipient chooses this number to be larger than any number assigned to messages currently queued or

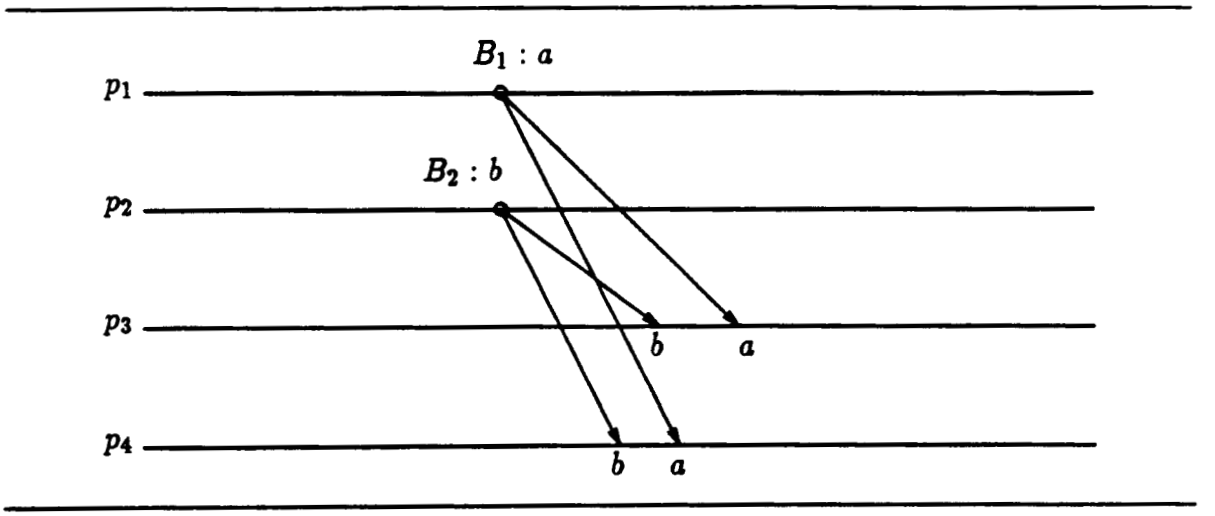


Figure 2.3: Atomic broadcast

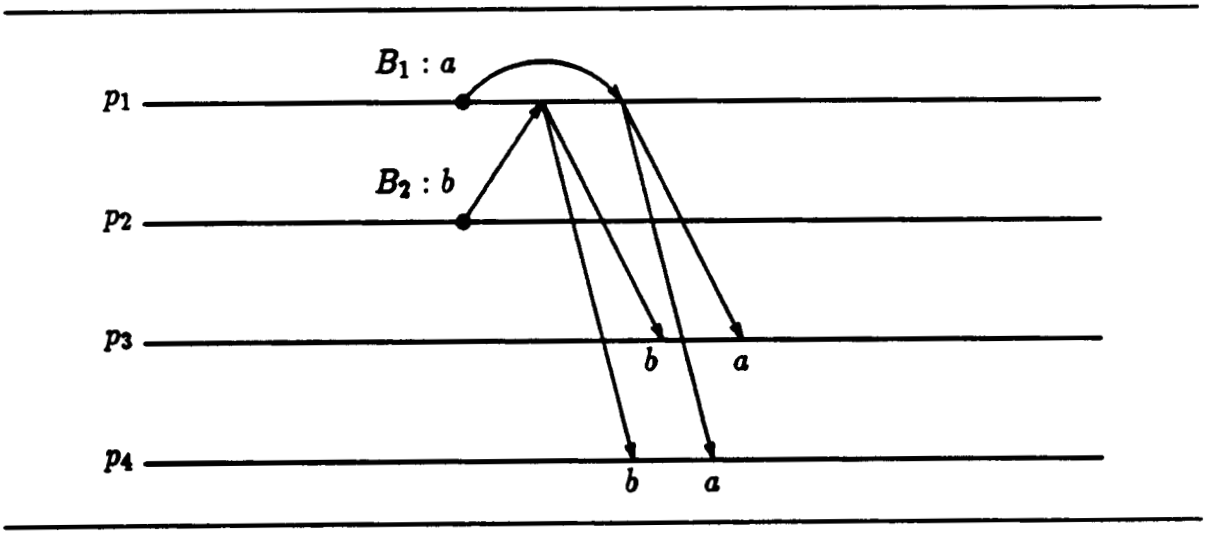


Figure 2.4: Two-phase implementation of atomic broadcast

previously delivered. The sender collects all priority numbers, computes their maximum and sends this number to all destination sites (phase three). Every recipient replaces the temporary priority number by the number just received, and reorders the queue accordingly. A message is delivered when it has received its final priority number and no messages with smaller priority number are in the queue.

Atomic ordering makes the design of fault-tolerant distributed applications much easier, because it reduces the uncertainty caused by message delays and failures in the system. However, this benefit does not come without cost. The two protocols described above need two or three phases of communication before an ABCAST message can be delivered. It is not difficult to prove that in an asynchronous system (i.e., a system with unbounded message delays), any protocol that guarantees atomic ordering requires some messages to take at least two hops before they are delivered. Consider for example a system with two processors, p_1 and p_2. Processor p_1 broadcasts a message a; at the same time p_2 broadcasts b. Both message are addressed to both processors. We claim that either message a needs at least two hops (to p_2 and back to p_1) before it can be delivered at p_1, or message b needs two hops. Assume the protocol delivers a at p_1 in one hop. This means that p_1 sends a to p_2, but it delivers the message locally without waiting for a reply from p_2 (See Figure 2.6). At the time of this local delivery, p_1 may not yet know that p_2 has sent a broadcast. If the message b from p_2 to p_1 is delayed long enough, the protocol will deliver a before b at p_1. Similarly, it is possible that at p_2, b will be delivered before a. But that would violate atomic ordering.

The situation is different if there is a known upper bound on message delays. In such a system it is possible to maintain synchronized clocks [BD87,LMS85,ST87]. In this case, atomic ordering can be achieved by a method based on *timestamps*. The

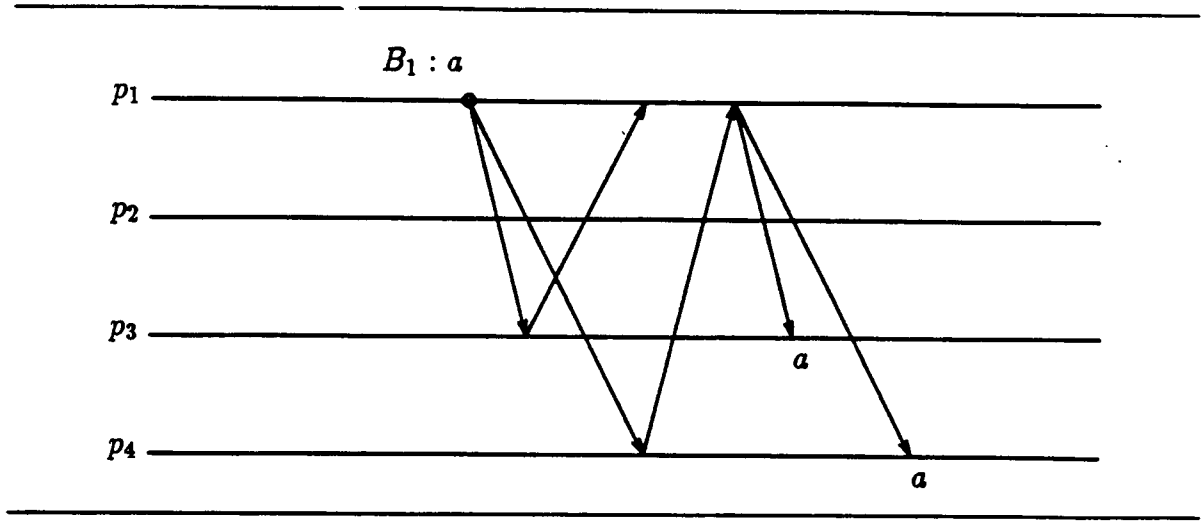


Figure 2.5: Three-phase implementation of atomic broadcast

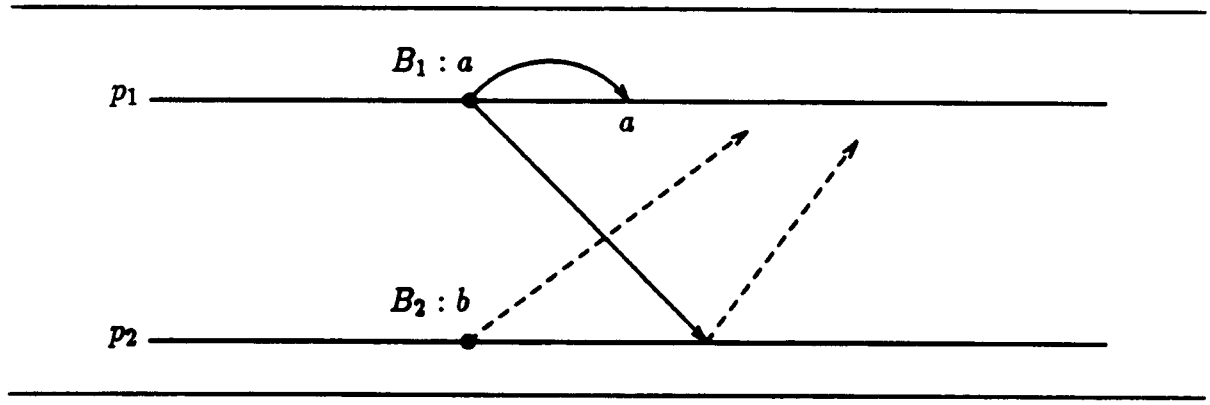


Figure 2.6: Processor p_1 delivers message a locally without waiting for any messages from p_2.

sender of a broadcast adds a timestamp to the message that shows the value of its local clock when the message is sent out. Messages received at a destination site are delivered to the application program in timestamp order; however, before a message is delivered, the processor has to wait until it is certain that no more messages with a lower timestamp will arrive. The amount of time to wait depends on the worst case message delay and on how closely clocks are synchronized [CASD84]. The disadvantage of this approach is that the delivery of *every* message is delayed by the worst case message delay, which is often much larger than the average delay in a two or three phase protocol.

2.1.4 Causal Broadcast

Because of the inherent cost of atomic broadcast protocols it is natural to look for protocols that provide stronger ordering than FBCAST but are less expensive than ABCAST. The *causal broadcast* (CBCAST for short) is such a protocol. It is based on the idea of *potential causality* introduced by Lamport in [Lam78].

The flow of information during the execution of a distributed system can be used to define a partial order on events occurring in the system. Such events are the *sending of a message*, the *receipt of a message*, or a *local event* that only affect a single processor. Figure 2.7 illustrates this. Events e_1, e_4, e_{11}, and e_{14} are *send-events*, $e_2, e_5, e_7, e_8, e_{12}, e_{13}$, and e_{15} are *receive-events*, and e_3, e_9, and e_{10} are *local-events*. According to Lamport's definition, all events that are connected by a path in this diagram are *potentially causally related*. Such a path must follow the horizontal lines (from left to right) or message arrows. For example, e_{10} is potentially causally related to e_1, because there is a path from e_1 to e_{10} going through e_2, e_4, and e_8 (dotted line in the figure). This dependency is denoted by the symbol "$\rightarrow$", i.e.,

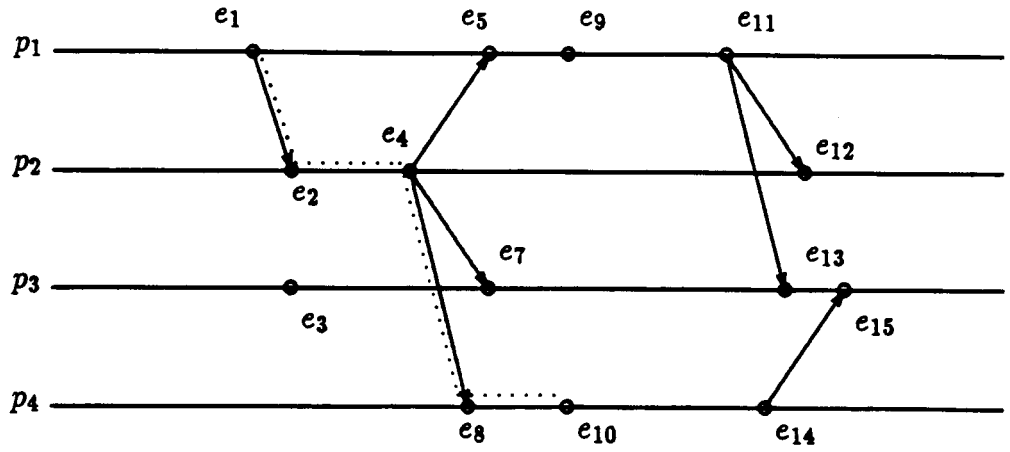


Figure 2.7: Potential causality

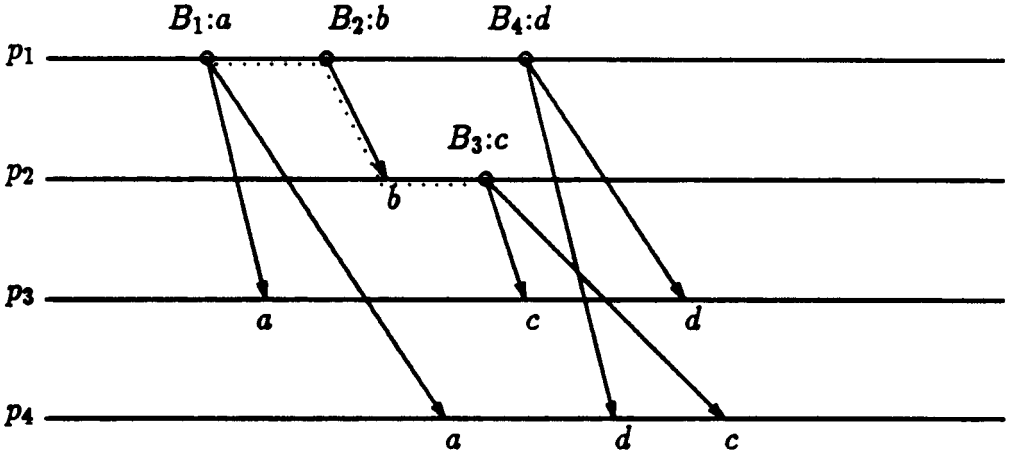


Figure 2.8: Causal broadcast

$e_1 \rightarrow e_{10}$. Events that are not connect by such a path are called *concurrent*. This is denoted by the symbol "//". For example, $e_5 // e_{14}$. The relation "$\rightarrow$" is called *potential causality* or *information flow relation*. The name "potential causality" has the following explanation. In physics, the Principle of Causality says that a cause has to precede its effect. Similarly, an event a in a distributed system can affect an event b at some other processors only if there is a flow of information from a to b, i.e., if a precedes b under "$\rightarrow$".

The ordering properties of a causal broadcast protocol are defined in terms of this information flow relation. CBCAST guarantees that every processor receives messages in an order that is consistent with "$\rightarrow$". That is, whenever two CBCAST send events are related by "$\rightarrow$", the protocol ensures that the two messages are received in the same order everywhere, namely the one given by "$\rightarrow$". For example, in Figure 2.8, broadcasts B_1 and B_3 are potentially causally related ($B_1 \rightarrow B_2$, represented by the dotted line in Figure 2.8). Consequently the message a is received before c at both p_3 and p_4. Broadcasts B_3 and B_4, on the other hand, are concurrent ($B_3 // B_4$). Hence the two messages c and d may be received in different orders at p_3 and p_4, as shown in this example. Notice that two events at the same processor are never concurrent. Therefore a causal broadcast also respects FIFO ordering. For example, in Figure 2.8, $B_1 \rightarrow B_4$; hence message a is received everywhere before d.

There are several ways of implementing causal broadcast that are very similar to the use of sequence numbers in FBCAST protocols. A processor wishing to broadcast a message adds some additional dependency information to the message before sending it to its destinations. This technique is called "piggybacking". The information that is added to an outgoing message m consists of a list of other, previously received messages that precede m under "$\rightarrow$". This form of CBCAST protocol is de-

scribed in detail by Birman and Joseph in [BJ87b]. In a system in which no failures occur, it is sufficient to transmit only message-ID's, instead of piggybacking whole messages onto other messages [Pet87]. Using this piggybacking technique causal ordering can be achieved without multiple phases of message exchanges.

2.2 Reliability

The broadcast protocols as we described them in the previous section only work correctly if no failures occur. Consider for example the BCAST protocol. If the sender crashes in the middle of the protocol, the message will reach only a subset of the destination sites. The situation is even worse for the three-phase ABCAST protocol. The failure of a single destination site can cause the protocol to block, preventing all other broadcasts from being received.

So-called *reliable* broadcast protocols avoid this undesirable behavior. A reliable broadcast guarantees that every message sent will eventually be received by all operational destination sites, despite processor failures. We have to qualify this statement a little. Under certain failure patterns, no protocol can guarantee the delivery of a broadcast to all operational destinations. For example, the sender could crash before it actually sent out any messages. Even if the sender manged to communicate with some other processor before it crashed, this other processor could experience a failure before talking to anybody else. In general, a set of failures in an early stage of a broadcast protocol could wipe out all knowledge about the message to be sent. What we mean by reliable message delivery is that a message is delivered to all operational destinations *unless* the sender fails before the protocol has terminated. Furthermore, in case the sender fails at some time during the protocol, message delivery must be *all-or-nothing*. More precisely:

If processor p sends a message m to a set D of destination sites, then the system will eventually reach one of the following two states:

1. For all $q \in D$: q has received m or q has crashed.
2. Processor p has crashed, and for all $q \in D$: q has crashed or q will never receive m.

This property is also called *atomic message delivery.*

We will now look at the different types of broadcast protocols introduced in the previous section (BCAST, FBCAST, CBCAST, ABCAST) and examine how they can be made reliable.

2.2.1 Reliable Bcast, Fbcast, and Cbcast

The simplest reliable broadcast protocol uses a method called *flooding* or *message diffusion.* A processor wishing to broadcast a message sends it to all destination sites by means of an (unreliable) BCAST. Every processor that receives the message forwards it to all other destination sites using BCAST. This way every destination will receive multiple copies of the message (one from the sender and one from each other destination, if no failures occur); it forwards the message only the first time it is received and ignores all duplicates. This protocol achieves atomic delivery: Every processor that receives the message will eventually either succeed in forwarding it to all other operational destinations or it will fail. Therefore, eventually either all operational sites have received the message or all sites that ever received the message have crashed.

By appending a set of sequence numbers to every message, FIFO ordering can be added to a diffusion protocol. This way we get a reliable FBCAST.

The same technique can be used to make CBCAST reliable. The CBCAST protocols we described in Section 2.1.4 work by piggybacking dependency information onto the broadcast message to be sent out. The use of message diffusion to propagate this message ensures that the original message contents as well as the dependency information are delivered to all operational destination sites. This way causal ordering can be preserved despite processor failures. For details see [BJ87b].

2.2.2 Reliable Abcast

ABCAST is a form of consensus protocol, because atomic ordering requires all processors to agree on total order on all broadcasts. In [FLP85] Fisher, Lynch, and Paterson show that it is impossible to achieve consensus in an asynchronous system if failures occur. Consequently, it is not possible to implement reliable atomic broadcast in such a system. The reason for this is that if no upper bound on message delays is known, a processor failure is indistinguishable from very slow communication. For example, consider a system with two processors p_1 and p_2. It is not difficult to prove the impossibility result of [FLP85] for this example: Assume processor p_1 broadcasts a message a with destination p_1, p_2. At the same time p_2 sends a message b, also addressed to both processors. Consider the following three scenarios:

1. Processor p_2 crashes before sending any messages; p_1 does not fail. Then the message a must eventually (say after some time interval d_1) be delivered at p_1.

2. Processor p_1 crashes before sending any messages; p_2 does not fail. Then the message b must eventually (say after some time interval d_2) be delivered at p_2.

3. Neither of the two processors fails, but the communication network is very slow; every messages takes at least time $d = max(d_1, d_2)$ before it is received.

Up to time d, processor p_1 cannot distinguish Scenario 3 from 1. In both cases it has not yet received any messages from p_2, but it does not know if p_2 has crashed or is still alive. Therefore, in Scenario 3, p_1 will deliver message a after time d_1, before receiving any messages from p_2. Similarly b will be delivered at p_2 before p_2 receives any messages from p_1. But then atomic ordering is violated in this scenario.

Therefore reliable atomic broadcast can only be achieved if we relax the assumptions about the asynchrony of the system. There are two ways of doing this:

1. Assume that failures can be detected. The ABCAST protocols described in [CM84] and [BJ87b] achieve reliability under this assumption. If a processor participating in an ABCAST protocol experiences a failure, some other processor can take over and complete the protocol on behalf of the crashed processor.

2. Assume there is an upper bound on message delays. In this case a reliable atomic broadcast can be implemented by combining a diffusion protocol with the method of timestamps to achieve atomic ordering [CASD84]. However, the amount of time that a processor has to wait before a message can be delivered to the application program increases with the number of expected failures.

Notice that the second assumption implies the first. If message delays are bounded, failures can be detected by timeouts. In fact, most failure detection mechanisms in distributed systems rely on timeouts.

2.3 Summary

We examined a variety of reliable broadcast protocols that differ in the form of message ordering they provide.

- **Atomic Broadcast** (ABCAST):

  All messages are delivered in the same order everywhere.

- **Causal Broadcast** (CBCAST):

  The order in which messages are delivered is consistent with the information flow relation between broadcast events.

- **Fifo Broadcast** (FBCAST):

  Broadcasts by the same processor are delivered in the order sent.

- **Unordered Broadcast** (BCAST):

  Messages are delivered in an arbitrary order.

The stronger the the ordering property of the broadcast, the more costly it is to implement. An atomic broadcast protocol requires at least two phases of message exchange, whereas CBCAST, FBCAST, and BCAST can be implemented as one-phase protocols. Furthermore, in an unreliable system in which processors may experience failures, ABCAST can only be implemented if failures are detectable or if an upper bound on message delays is known.

Chapter 3

Formal Model

In this chapter we present a formalism based on events and histories for specifying problems in a distributed system. We introduce a model for a broadcast-based distributed implementation and give a definition for the correctness of an implementation with respect to a problem specification. We illustrate our formalism by showing that every formal problem has an implementation based on atomic broadcasts.

3.1 Formal Problem Specifications

A program running in a distributed system consists of several components, each running at a different site, and interacting with each other by sending and receiving messages. A formal specification for such a program can be given in terms of its *input/output* behavior. At each site there are clients (human users or other programs) that interact with the distributed program. This interaction is typically described by a procedural interface. A client invokes an operation by passing the operation name and a set of parameters to the component of the distributed program

residing at the local site. The program executes the operation, informs the client of its completion, and possibly returns a value to the client. During the execution of the operation the local component of the program may interact with remote components of the program. Figure 3.1 illustrates this view of a distributed program.

We distinguish between the implementation of a distributed program and its behavior as observed by its clients. From a client's point of view the program is a *service* that accepts requests from clients at different sites, executes each request and returns the result to the client. Figure 3.2 illustrates this view of a distributed program as a centralized service. We use this client view as the basis of our formal specifications.

Definition 3.1

A formal event

$$e = A_i(x_1, \ldots, x_n) : v$$

denotes operation A invoked by client i with parameters $x_1, \ldots, x_n$, and returning the value v. A formal history

$$H = (e_1, e_2, \ldots, e_m)$$

is a finite, totally ordered sequence of events.

A formal history describes the sequence of operations executed by the service and the values returned to the clients. A *formal specification* determines what constitutes correct behavior of the service, by defining which formal histories of the service are *legal*. Since we do not want to commit ourselves to any particular logical language for describing specifications, we simply identify a specification with the set of histories accepted by it.

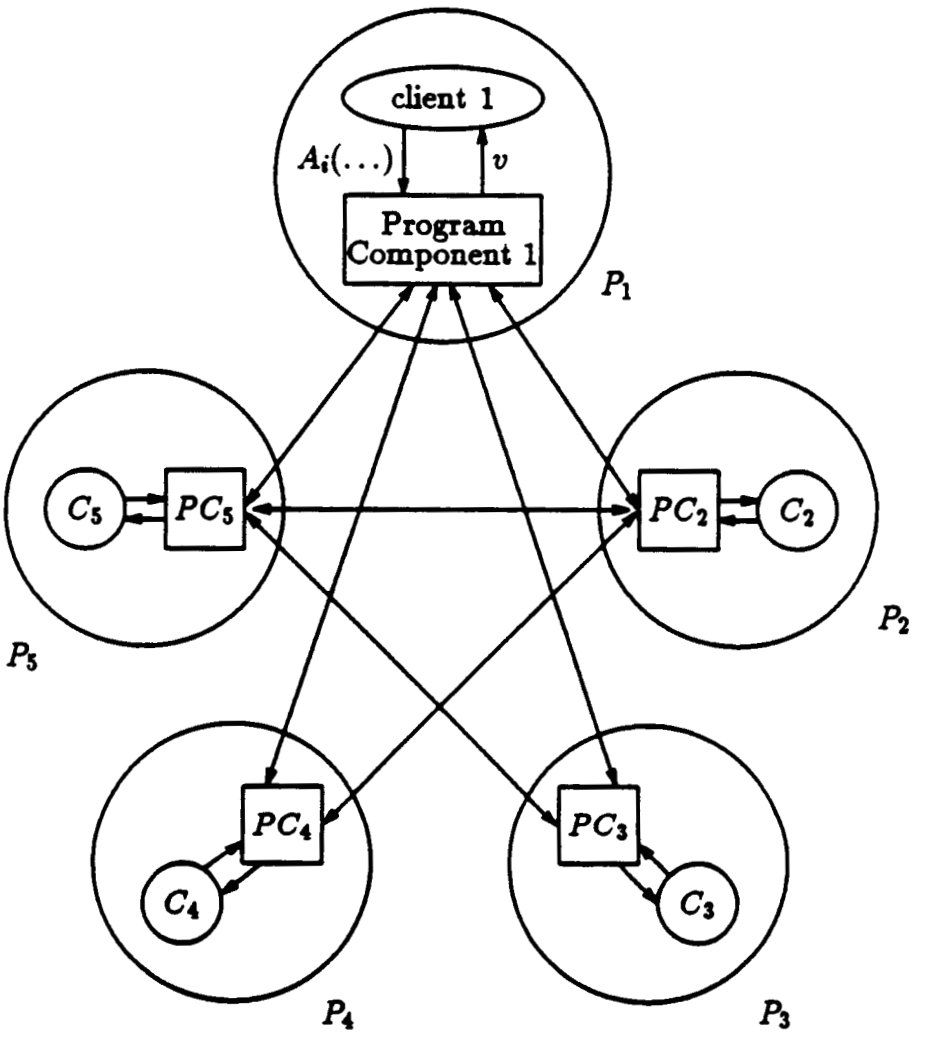


Figure 3.1: A client interacting with a distributed program

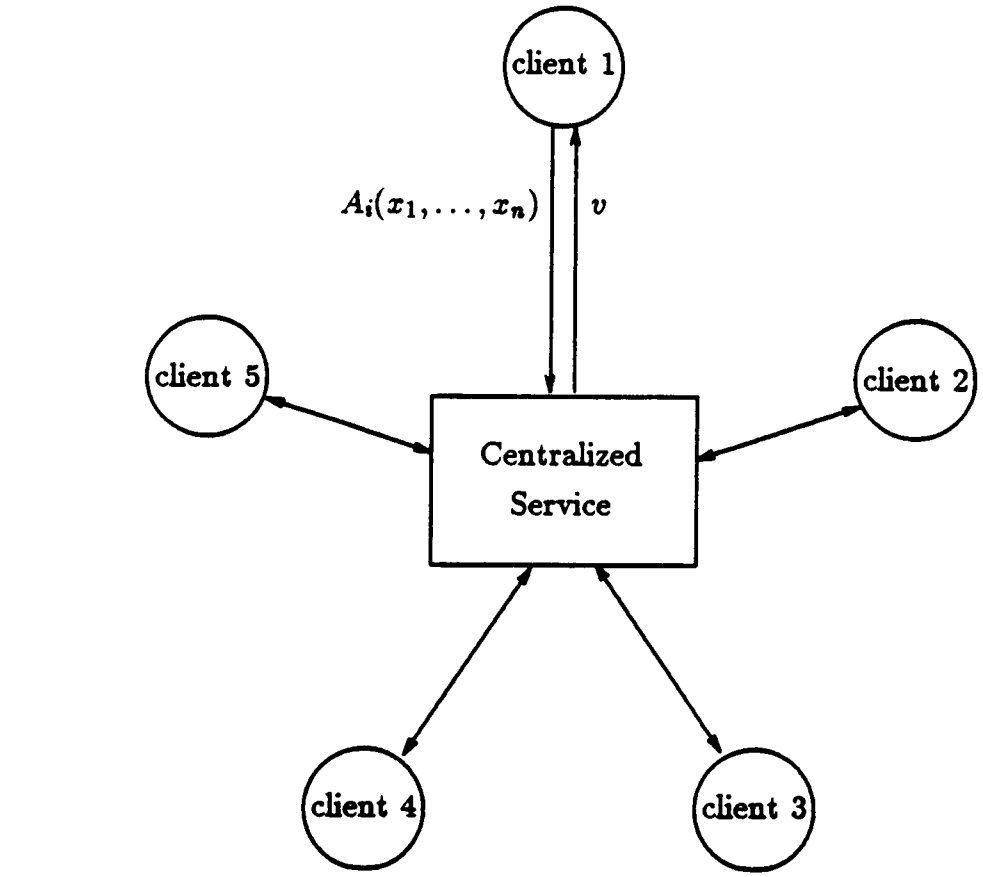


Figure 3.2: Client view of a distributed program

Definition 3.2

$H+e$ or He	denotes the history obtained by appending the event e to H.
HH'	denotes the concatenation of H and H'.
$H \leq H'$	means that H is a prefix of H'.
$H\|_i$	denotes the projection of H onto client i, that is the subsequence of H containing all operations invoked by client i.

Definition 3.3

A formal specification is a quadruple $S = (n, I, V, S)$, where n is the number of clients, I is a set of invocations of the form $A_i(x_1, \ldots, x_k)$, V is a set of return values, and S is a set of histories. S must satisfy the following two properties:

S is prefix-closed: $\forall H \in S: \ \forall H' \leq H: \ H' \in S$,

S is complete and deterministic:

$\forall H \in S: \ \forall$ invocation $a \in I:$

$\exists$ unique return value $v \in V: \ H + a{:}v \in S.$

At this point it is useful to give an example that illustrates our formalism. We will use this example throughout the rest of this dissertation. Consider the problem of managing a shared resource in a distributed system. The resource can be accessed from any site, but we want to ensure that at any given time only one site actually uses the resource. This problem can be solved by introducing the concept of a *token* that is associated with the resource. Only the site that is currently holding the token is allowed to access the resource. If the current token holder no longer needs the resource it may *pass* the token to some other site. We want to design a token

passing service that manages this token. This service would support the following operations:

- QUERY(): BOOLEAN

  — *returns* TRUE *if the caller is the current token holder.*

- PASS(X: CLIENTID): RETURNCODE

  — *passes the token from the current token holder to client* x.

  The PASS operation returns one of three values: OK, ERRORHOLDER (the caller is not the current token holder), or ERRORREQUEST (client x did not request the token).

- REQUEST(): RETURNCODE

  — *request the token.*[1]

  The REQUEST operation returns one of three values: OK, ERRORHOLDER (the caller is already holding the token), or ERRORREQUEST (the caller has already requested the token).

A complete formal specification is given in Appendix A. Here we will only list a few histories that illustrate this example.

We assume that initially the token is held by client 1. Consider the history H_1:

$$H_1 = \ Q_3{:}F,\ R_3{:}ok,\ P_1(3){:}ok,\ Q_3{:}T$$

Q, R, P stand for QUERY, REQUEST, and PASS operations; T and F stand for the return values TRUE and FALSE. Client 3 invokes a QUERY and finds out that it is

[1]The request operation is non-blocking. A client that needs the token would invoke a request operation and then repeatedly issue a QUERY operation until it returns TRUE.

not holding the token. It then decides to request the token. Client 1 (the initial token holder) passes the token to client 3, and consequently a QUERY by client 3 returns TRUE. We would consider this a legal history, i.e., $H_1 \in S$.

$$H_2 = \ Q_3{:}F,\ R_3{:}ok,\ P_1(3){:}ok,\ Q_3{:}F$$

H_2 is an example of an illegal history: although the token has been passed to client 3, the last QUERY returns FALSE. In this example the token passing service would have returned the wrong value for the QUERY; therefore $H_2 \notin S$.

$$H_3 = \ Q_3{:}F,\ R_3{:}ok,\ P_3(2){:}ok,\ Q_3{:}F$$

This history is also illegal: client 3 is passing the token although it is not holding it. The token passing service behaved incorrectly by returning OK for this operation. It should instead have returned the value ERRORHOLDER, indicating an error:

$$H_3 = \ Q_3{:}F,\ R_3{:}ok,\ P_3(2){:}ErrorHolder,\ Q_3{:}F$$

In our formalism we make a number of implicit and explicit assumptions about the distributed service.

1. A client invokes only one operation at a time and waits for the operation to complete before invoking the next one.

2. We assume every operation can be executed as soon as it is invoked. In particular there are no operations that explicitly wait until another client has taken a certain action. In our formalism, operations with wait semantics must be modeled by a "busy wait". The token passing service for example does not have an operation WAITFORTOKEN. Instead we provide the REQUEST and QUERY operation. A client waiting for a token would periodically invoke a QUERY until it returns TRUE. In Appendix B we show formally that any

operation with wait semantics can be modeled in this way. We chose this model because it is simpler and entails no loss of generality.

3. We require specifications to be prefix-closed. This allows us to decide, at any time during the execution of a system, whether the service has behaved correctly so far. In other words, the correctness of an execution up to a given time does not depend on any future events. The prefix closure of S also makes it unnecessary to consider infinite histories. An infinite, legal history is represented in S by all its (finite) prefixes. However, because histories are finite and specifications are prefix-closed, our formalism can only express safety properties, not liveness properties [SA85].

4. We only consider deterministic specifications in which the value returned by an operation is determined completely by the parameters of the operations and by the previous history. Also, because specifications are complete, all operations are *total*. In other words, clients are not restricted to invoke only "legal" operations. Any specification can be made complete by specifying that an operation should return a distinguished value ERROR when performed in a state in which it would otherwise not be legal to execute the operation.

Our specifications differ from other formal specification methods. In particular, we do not associate any *state variables* with a service. For example, consider a service that provides two operations READ and WRITE. Instead of saying that a WRITE(X=5) changes the value of some internal variable x, we specify the effect of this operation by saying that the next operation READ(X) should return the value 5. Rather than specifying how an operation changes the internal state of a service, we specify how the operation affects the result of future operations.

In some sense these two approaches for formal specifications are equivalent. In our formalism the current state of a service is represented by the history of all operations executed so far. A new operation changes this state by appending an event to the history. We chose the history-based approach because it does not assume any specific internal representation for the state of the service.

3.2 System Execution Model

The main goal of this dissertation is to find out how different forms of broadcasts can be used to construct a solution to a problem that is specified in the formalism we introduced in the previous section. In this section we present a model for studying broadcast-based implementations of a service.

In the most general terms, a distributed implementation of a service runs like this:

- A client at processor i invokes an operation a.
- Processor i starts an *agreement protocol* among all processors to decide on the effect of the operation and its return value.
- When the protocol terminates, the result is returned to the client.

We will show in Section 3.5 that in order to obtain an implementation of any formally specified problem, it is sufficient to have agreement protocol establish a global order on all the operations invoked by different clients in the system. An atomic broadcast (ABCAST) does just that. An implementation based on ABCAST would run like this:

- A client at processor i invokes an operation a.

- Processor i puts the operation (including its parameters) into a message and broadcasts the message to all sites in the system (including itself).
- Other processors that receive this message update their local state.
- When site i receives its own message, it also updates its state and at that time computes the result to be returned to the client.

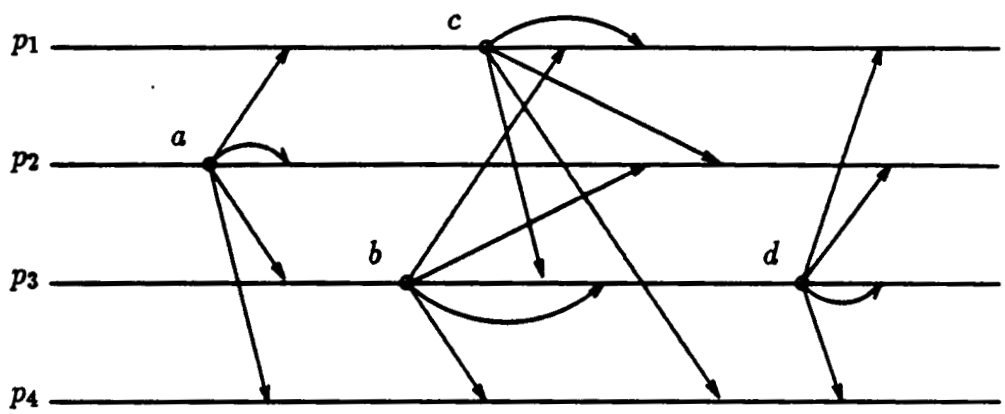
In Section 3.5 we will make this more precise and prove that such an implementation indeed gives a correct solution for any specification. In Chapter 4 we will then explore conditions under which it is possible to replace the ABCAST by a more efficient broadcast protocol that does not require the client to wait for a multi-phase agreement protocol to finish before it gets back its return value. In the rest of this section we define our model for broadcast-based implementations and give a criterion for their correctness with respect to a formal specification.

3.2.1 Execution Histories

An execution of a broadcast-based implementation outlined above can be described by a picture like Figure 3.3. The horizontal lines show events happening at different processors. To simplify the model we assume that there is only one client per processor. There are two different types of execution events[2].

1. *Invocation events*, which denote the invocation of an operation by the local client. An invocation causes a message to be broadcast to all sites. These messages are represented by the arrows in the figure.

[2]Note that execution events are different from formal events as in Definition 3.1. Definition 3.11 in Section 3.2.2 relates these two types of events.



$$
\begin{aligned}
E_1 &= (2,1)\ \ c\ \ (3,1)\ \ (1,1)\ \ (3,2) \\
E_2 &= a\ \ (2,1)\ \ (3,1)\ \ (1,1)\ \ (3,2) \\
E_3 &= (2,1)\ \ b\ \ (1,1)\ \ (3,1)\ \ d\ \ (3,2) \\
E_4 &= (2,1)\ \ (3,1)\ \ (1,1)\ \ (3,2)
\end{aligned}
$$

Figure 3.3: An execution history

2. *Receive events*, which denote the receipt of a message that was broadcast from some other site. The tip of each arrow represents a receive event in the figure.

Consider, for example, the events at processor 2 in Figure 3.3. The first event is an operation "a" invoked by the client at that site. The invocation causes the processor to send out a broadcast, which is represented by the four arrows originating from the circle at "a". The end of an arrow represents the receive event that arises when the broadcast message is delivered at another (or the same) site. A receive event is labeled by a pair of integers; the first one designates the processor that sent the broadcast, and the second one counts broadcasts sent from that processor. The second event at p_2 is the receive event (2,1), denoting the delivery of the first broadcast from itself. There are three more receive events at p_2: (3,1), (1,1) and (3,2). They denote the delivery of the first broadcast from p_3, the first one from p_1, and the second broadcast from p_3. Below we describe such a graphical representation of an execution in formal terms:

Definition 3.4

An execution sequence $E = (E_1, \ldots, E_n)$ is a collection of totally ordered sets of *invocation* and *receive* events,

$$E \in [(I \cup N^2)^*]^n,$$

satisfying the condition:

$$\forall\, inv_E\langle i,j\rangle\colon\ \forall\, k\colon\ \exists \text{ unique receive event } (i,j) \in E_k,$$

where $inv_E\langle i,j\rangle$ denotes the j'th *invocation* event in E_i.

We now introduce some terminology and notation:

Definition 3.5

F_i	the j'th event in E_i.
$<_i$	the order of events in E_i, i.e., $E[i,j] <_i E[i,j']$ iff $j < j'$.
$inv_E\langle i,j\rangle$	the j'th *invocation* event in E_i.
$rcv_E(\langle i,j\rangle,k)$	the receive event (i,j) in E_k.
$inum_E\langle i,j\rangle$	the sequence number of $inv_E\langle i,j\rangle$ in E_i, i.e., if $E[i,l] = inv_E\langle i,j\rangle$ then $inum_E\langle i,j\rangle = l$.
$rnum_E(\langle i,j\rangle,k)$	the sequence number of $rcv_E(\langle i,j\rangle,k)$ in E_k. i.e., if $E[k,l] = rcv_E(\langle i,j\rangle,k)$ then $rnum_E(\langle i,j\rangle,k) = l$.
$E - a$	(where $a = inv_E\langle i,j\rangle$ is an invocation event) the execution history that is identical to E except that a and all its corresponding receive events ($rcv_E(\langle i,j\rangle,k)$, for all k) are deleted.

Definition 3.6

Given an execution sequence E we define the relation "$\xrightarrow{D}$" on the events in E:

$$E[i,j] \xrightarrow{D} E[i,j+1] \qquad \text{for all } i,j.$$

$$inv_E\langle i,j\rangle \xrightarrow{D} rcv_E(\langle i,j\rangle,k) \qquad \text{for all } i,j,k.$$

If $a \xrightarrow{D} b$ we say that a directly precedes b.

An execution like the one in Figure 3.3 can be viewed as a directed graph that has invocation and receive events as nodes and two types of edges: the horizontal lines that connect events happening at the same processor and the arrows that represent broadcast messages. The "$\xrightarrow{D}$" relation defines the edges in the execution graph.

For an execution sequence to make sense we need to add a few more restrictions to Definition 3.4. For example, we need a condition that prevents messages from flowing backwards in time.

Definition 3.7

An execution history $E = (E_1, \ldots, E_n)$ is an execution sequence satisfying the following additional conditions:

- **Sequential invocation:** Clients invoke operations sequentially, i.e., a client waits for the present invocation to complete before invoking a new one:

$$\forall i, j: \quad rcv_E(\langle i, j \rangle, i) <_i inv_E(i, j+1\rangle$$

- **Monotonicity of time:** The "$\xrightarrow{D}$" relation is acyclic (messages do not flow backwards in time):

$$\neg\, \exists\, e_1, \ldots, e_m \in E: \quad e_1 \xrightarrow{D} e_2 \xrightarrow{D} \ldots \xrightarrow{D} e_m \xrightarrow{D} e_1.$$

In addition we may specify the ordering properties of the broadcast protocol used by giving a *message ordering axiom*. For example, if we are interested in systems in which an atomic broadcast is used, we would specify an ABCAST-axiom that ensures that all messages are received in the same order everywhere:

Definition 3.8

ABCAST ordering axiom:

$\forall i, j, i', j':\ \forall k, l:$

$$rcv_E(\langle i, j \rangle, k) <_k rcv_E(\langle i', j' \rangle, k) \quad \Leftrightarrow \quad rcv_E(\langle i, j \rangle, l) <_l rcv_E(\langle i', j' \rangle, l).$$

An execution history that satisfies this ABCAST axiom in addition to the requirements of Definition 3.7 would be called an "ABCAST execution history".

3.2.2 Implementations

The previous section described a system execution only in terms of what operations clients invoke and when messages are sent and received. It does not specify what the contents of these messages are, how the recipient processes such a message, or what values are returned to the client as the result of an invocation. In other words, we need to specify what the program running at each site does.

We do this by modeling each processor as a state machine that reacts to input events (invocation events or receive events) by changing its state and generating an output event (message to be broadcast or value returned to the client). This state machine has two types of transition functions (ϕ and ψ), corresponding to the two types of input events.

Definition 3.9

An implementation is a 8-tuple $(n, I, V, M, Q, q_0, \Phi, \Psi)$, where

n	the number of processors in the system
I	the set of operations that can be invoked
V	the set of return values
M	the set of message values
Q	the set of states in which a processor can be
q_0	the initial state of all processors
$\Phi = (\phi_1, \ldots, \phi_n)$	invocation transition functions $\phi_i : Q \times I \rightarrow Q \times M$
$\Psi = (\psi_1, \ldots, \psi_n)$	message receive transition functions $\psi_i : Q \times M \rightarrow Q \times V$

The meaning of the transition functions ϕ_i and ψ_i is as follows: When an operation $a \in I$ is invoked by client i, processor i changes its state from q to q' and broadcasts the message m, where $(q', m) = \phi_i(q, a)$. When such a message is received at site j, processor j changes its state from q to q' , where $(q', v) = \psi_j(q, m)$. The return value for this operation is v; at the site where the operation was invoked this value is passed to the client; at the other sites it is ignored. We will use superscripts s, m, v to refer to the state, message, and return value of ϕ and ψ, respectively, as defined below:

$$\text{if} \quad \phi_i(q,a) = (q',m) \quad \text{then} \quad \phi_i^s(q,a) = q', \quad \phi_i^m(q,a) = m;$$
$$\text{if} \quad \psi_i(q,m) = (q',v) \quad \text{then} \quad \psi_i^s(q,m) = q', \quad \psi_i^v(q,m) = v.$$

Given such a formal implementation, we can take an execution history and determine what messages are sent and what values are returned to the client. We start by giving a definition for computing the state of a processor after a particular event in an execution history:

Definition 3.10

$$stat_E[i,j] = \begin{cases} q_0 & \text{if } j = 0 \\ \phi_i^s(stat_E[i,j\text{-}1], a) & \text{if } E[i,j] = a \text{ is an invocation event} \\ \psi_i^s(stat_E[i,j\text{-}1], m) & \text{if } E[i,j] = (k,l) \text{ is a receive event, where} \\ & m = \phi_k^m(stat_E[k, inum_E\langle k,l\rangle\text{-}1], inv_E\langle k,l\rangle) \end{cases}$$

Then $stat_E[i,j]$ defines the state of processor i after $E[i,j]$, the j'th event at that site. Note that the monotonicity requirement for execution histories (Definition 3.7) prevents this definition from being circular. It is now straightforward to give definitions that compute the messages being sent, the values returned to clients, the

formal events (invocation plus return value as in Definition 3.1) observed by clients, as well as the sequence of formal events that any particular client observes:

Definition 3.11

$msg_E\langle i,j\rangle = \phi_i^m(stat_E[i, inum_E\langle i,j\rangle - 1], inv_E\langle i,j\rangle)$

$val_E\langle i,j\rangle = \psi_i^v(stat_E[i, rnum_E(\langle i,j\rangle, i) - 1], msg_E\langle i,j\rangle)$

$event_E\langle i,j\rangle = a{:}v$, where $a = inv_E\langle i,j\rangle$, and $v = val_E\langle i,j\rangle$.

$H[E,i] = (event_E\langle i,1\rangle, event_E\langle i,2\rangle, \ldots, event_E\langle i,m\rangle)$,
where m is the number of invocation events in E_i.

3.3 Implementation Correctness

In Section 3.1 we defined formal problem specifications in terms of totally ordered histories which record a sequence of events executed by a centralized service. In our model of distributed implementations, however, there is no centralized service. Instead of one global history, we have a set of histories $H[E,i]$ containing the subset of events observed by individual clients. We consider such a distributed implementation correct if, to the clients, its behavior is indistinguishable from the behavior of a centralized service which performed the same set of operations. In particular, the implementation must satisfy the following condition:

> For every execution history E, it must be possible to merge all $H[E,i]$ into one legal, global history $H \in S$.

This ensures that clients cannot distinguish an execution of the distributed implementation from a centralized one, because they all see part of a history that

would have been generated by a centralized server. This correctness condition is very similar to the notion of serializability familiar from database theory [BG81, Pap79].

This condition alone is not enough to ensure that the distributed implementation behaves as one would expect. We need to add a condition that says something about the relative order in which events invoked at different processors appear in the global history H. Consider the token passing example from Section 3.1. One could implement the token service in the following trivial way (recall that client 1 is the initial token holder):

- QUERY always returns FALSE if it is invoked by any client other than client 1.
- If invoked by client 1, QUERY returns TRUE until client 1 passes the token. After the event $P_1(j) : ok$ a QUERY by client 1 always returns FALSE.

This implementation effectively "loses" the token after the first pass operation, because subsequent queries by any client return FALSE. However, notice that the implementation satisfies the correctness condition stated above. An execution history for this service might generate the following collection of formal histories observed by clients:

$$H[E,1] = \ Q_1{:}T,\ P_1(3){:}ok,\ Q_1{:}F,\ Q_1{:}F$$
$$H[E,2] = \ Q_2{:}F,\ Q_2{:}F, Q_2{:}F,\ \ldots,\ Q_2{:}F$$
$$H[E,3] = \ Q_3{:}F,\ Q_3{:}F, Q_3{:}F,\ \ldots,\ Q_3{:}F$$

These three histories can easily be merged in to a legal history:

$$H = \ Q_1{:}T,\ Q_2{:}F,\ Q_3{:}F,\ \ldots,\ Q_1{:}T,\ Q_2{:}F,\ Q_3{:}F,\ P_1(3){:}1,\ Q_1{:}F,\ Q_1{:}F$$

In other words, by putting the PASS event (and everything following it in $H[E,1]$) at the very end of H, we always get a legal merged history.

To solve this problem we need to add a condition that prevents events from being indefinitely deferred in the merged history H. An event observed by one client should eventually get a "stable" place in H. We have to define what we mean by "eventually", since our execution model does not contain real time. Recall that we are assuming an asynchronous distributed system in which messages may be delayed arbitrarily. It could be that the broadcast protocol initiated for the PASS operation terminates quickly at site 1, whereas due to message delays it finishes much later at sites 2 and 3. In this case we would consider the execution outlined above an acceptable behavior of a distributed implementation. Therefore we add the following condition:

> Once a broadcast message about an event a has been received at site i, the event becomes "stable" with respect to other events at site i. That is, when we construct a legal, global history H by merging individual processor histories, any event b that was invoked at site i after the message about a was received at i, must be ordered after a in H.

In other words, we allow an event invoked at another site to be ignored only as long as the message about it is still in transit. In the token passing example above, this condition says that as soon as the message about the operation $\text{PASS}_1(3)$ is received at site 3, the next QUERY operation should return TRUE.

The next definition summerizes our two correctness conditions for distributed implementations.

Definition 3.12

Y is a correct XBCAST-implementation of specification $S = (n, I, V, S)$ iff:

$\forall$ XBCAST execution history E: $\exists H \in S$:

Correctness: $\forall i$: $H|_i = H[E, i]$

Liveness: $\forall i, j, k$:

$$rcv_E(\langle i, j\rangle, k) <_k inv_E\langle k, l\rangle \Rightarrow event_E\langle i, j\rangle <_H event_E\langle k, l\rangle$$

Here "XBCAST" stands for the type of broadcast used in the implementation. As discussed above, the second condition (liveness) makes sure that as long as the broadcast protocol guarantees that every message will eventually be be delivered everywhere, every operation invoked by a client will eventually be reflected in operations at other sites. In other words, liveness of the broadcast protocol implies liveness of the implementation.

3.4 Externally Observed Histories

Our model of execution histories does not contain any notion of *real time*. This raises the question: How does an execution history relate to what an external observer sees during the execution of an implementation? In an asynchronous system it does not make much sense to talk about time in absolute terms (e.g., milliseconds). However, we can consider the relative order — in real time — of events occurring during the execution of an implementation. Imagine an external observer who is able to monitor all nodes in a distributed system simultaneously. Such an observer would be able to determine a total order on all invocation and receive events at all sites. We call this sequence of execution events an *external history*, E_{ext}. We can make the following statement about the relationship between the formal execution

history E and the corresponding external history E_{ext}:

1. The formal execution history E already determines a total order on all events that happen at the same processor. Therefore, for all i, the events in E_i appear in exactly the same order in E_{ext}.

2. The relative order of events at different processors is not determined by E, except that a receive event can never precede its corresponding invocation event, because messages do not flow backwards in time.

We can summarize this in the following statement:

> The external history E_{ext} can be any total order on the events in E that is consistent with "$\xrightarrow{D}$".

Given E_{ext} we can extract an external formal history H_{ext} recording all the formal events during the execution of an implementation in the order they are seen by the external observer. Notice that our correctness definition does not imply that H_{ext} is always legal. However, it ensures that there exists a legal history that is *similar* to H_{ext}, as defined below.

Definition 3.13

H is <u>similar</u> to H' ($H \approx H'$) iff $\forall i$: $H|_i = H'|_i$

If clients communicate only through requests to the distributed service, then similar histories are indistinguishable to all clients. For a correct implementation it will always be the case that

$\forall$ execution history E: $\exists H \in S$: $H_{ext} \approx H$

For example, an event a may logically be ordered before b in H ($a <_H b$), but physically a could be observed after b, if a and b are events at different processors.

Hence the above statement just rephrases the requirement that a correct, distributed implementation be *indistinguishable* from a centralized implementation in which the externally observed history H_{ext} is always legal.

3.5 Abcast Implementation

In Section 3.2 we claimed that the strong ordering properties of the atomic broadcast provide enough synchronization between processors to solve any problem that can be specified in our formalism. In this section we will prove this claim. The purpose of this exercise is twofold. By showing that every problem has an ABCAST implementation, we demonstrate that our model of broadcast-based implementation is not too restrictive. Furthermore, in the next chapter we will use methods similar to the ones in this section to construct implementations based on more efficient protocols.

Given a formal specification $S = (n, I, V, S)$ we will construct an implementation that satisfies our Definition 3.12 of correctness for all ABCAST execution histories. Figure 3.4 describes this implementation informally in Pascal-like pseudocode. The implementation is essentially a variation of the state machine approach to replication as described in [Sch86]. The current system state is represented by the sequence of all operations executed so far (variable 'H' in Figure 3.4). This state, as well as the execution of client requests, is fully replicated. An operation invoked by a client is broadcast to every site (including the one at which it was invoked) and is executed everywhere when it is received. Executing an operation in a state H simply means adding a new event to H after choosing a suitable return value v, such that the new history is still legal. The requirement that specifications be deterministic and complete ensures that there is always exactly one choice for such a value. This,

Processor i runs the following program:

H := empty;
loop
    wait for an invocation by the local client or the receipt of a broadcast;
    if client invoked operation a **then**
        ABCAST "a" **to all processors;**
    else if broadcast "a**" was received from** j **then**
        pick a value v**, such that** $H + a : v \in S$;
        $H := H + a : v$;
        if $j = i$ **then return value** v **to the client end if**
    end if
end loop

Figure 3.4: ABCAST **implementation**

and the fact that ABCAST delivers all broadcasts in the same order at every site, implies that all processors will agree on the same legal history H of events that have occurred in the system. This history will satisfy the correctness condition in Definition 3.12. In order to prove this, we translate this implementation into our formal execution model.

Because specifications are deterministic and complete, we can define an *execution function* $\chi_S : S \times I \rightarrow V$ such that

$$\forall H \in S, a \in I_S: \quad v = \chi_S(H, a) \quad \Rightarrow \quad H + a{:}v \in S,$$

or in words: χ_S computes the correct return value of operation a invoked in state H. Given a specification $S = (n, I, V, S)$ we define the implementation Y_S:

$$Y_S = (n, I, V, I, (I \times V)^*, \emptyset, \Phi, \Psi), \quad \text{i.e., } M = I,\ Q = (I \times V)^*,\ q_0 = \emptyset,$$

where the transition functions Φ, Ψ are defined as follows: When operation a is invoked at processor i in state H, it does not change its state but broadcasts "a":

$$\phi_i(H, a) = (H, a).$$

When processor i receives a message containing operation a it executes the operation by adding the event $a{:}v$ to its history H; the value v is returned to the client:

$$\psi_i(H, a) = (H + a{:}v, v), \text{ where } v = \chi_S(H, a).$$

Lemma 3.1

For every ABCAST execution history E of Y_S: the final state of all processors is identical.

Proof: A processor state only changes when a message is received (ϕ_i^q is the identity function). Because of the ABCAST ordering axiom (Definition 3.8), all E_i

contain the same sequence of receive events. Since all processors start in the same state $q_0 = \emptyset$ and the transition functions ψ_i are identical, all processors will end up in the same final state. □

Theorem 3.1

Y_S is a correct ABCAST implementation of specification S.

Proof: We show that for every execution E, the history H_f given by the final state of processors in E is legal ($H_f \in S$) and satisfies the correctness and liveness conditions of Definition 3.12. We do this by induction on the number of events in H_f.

The base case, $H_f = \emptyset$, is trivially satisfied because an empty history is always legal. This follows from the fact that specifications are prefix-closed.

For the induction step, consider an execution history E such that H_f is non-empty. Let $rcv_E(\langle i,j\rangle, 1)$ be the last receive event in E_1. Because of the ABCAST ordering, $rcv_E(\langle i,j\rangle, k)$ is the last event in E_k for all k. Let $E' = E - inv_E\langle i,j\rangle$ (i.e., E with $inv_E\langle i,j\rangle$ and $rcv_E(\langle i,j\rangle, k)$, for all k, deleted). Let H'_f be the history given by the final state of processors in E'.

We first show that H_f is legal. By induction hypothesis $H'_f \in S$. Furthermore, $H_f = H'_f + a{:}v$, where

$$v = val_E\langle i,j\rangle = \psi_i^v(H'_f, a) = \chi_S(H'_f, a).$$

Therefore $H_f = H'_f + a{:}v \in S$ follows from the definition of χ_S. We complete the proof by showing that H_f satisfies the correctness and liveness conditions of Definition 3.12.

Correctness: We have to show that $H_f|_i = H[E,i]$ for all i. By induction hypothesis $H'_f|_i = H[E',i]$. Therefore

$$H_f|_i \;=\; (H'_f + a{:}v)|_i \;=\; H'_f|_i + a{:}v \;\;=\; H[E',i] + a{:}v \;\;=\; H[E,i].$$

Liveness: Let $rcv_E(\langle l,m\rangle,k) <_k inv_E\langle l',m'\rangle$. We have to show that $event_E\langle l,m\rangle < event_E\langle l',m'\rangle$ in H_f. **Case 1** $\langle l',m'\rangle = \langle i,j\rangle$: In this case $event_E\langle l',m'\rangle = event_E\langle i,j\rangle$ is the last event in H_f, and therefore $event_E\langle l,m\rangle < event_E\langle l',m'\rangle$ in H_f. **Case 2** $\langle l',m'\rangle \neq \langle i,j\rangle$: In this case the two events $event_E\langle l,m\rangle$ and $event_E\langle l',m'\rangle$ are both in H'_f, and by induction hypothesis $event_E\langle l,m\rangle < event_E\langle l',m'\rangle$ in H'_f, hence also in H_f. □

3.6 Summary

We presented two different models for a distributed program: one for the formal specification of the program and one for its implementation.

1. We modeled the behavior of a distributed program as a service that executes requests on behalf of clients. An execution of such a service is described as a sequence of events, in which each event denotes the execution of one client request. We call such an event sequence a formal history. A formal specification for such a service is a set S that lists all possible legal histories.

2. Our implementation model describes a system as a collection of state machines. Each processor reacts to input events (invocation events or receive events) by changing its state and generating an output event (message to be broadcast or value returned to the client).

We then defined the correctness of an implementation with respect to a formal specification in such a way that clients cannot distinguish the behavior of the distributed implementation from that of a central server.

To illustrate our formalism and to show that our implementation model is not too restrictive, we demonstrated that any formal specification has an ABCAST implementation.

Chapter 4

Asynchronous Implementations

In the previous chapter we saw that every formal specification has an ABCAST implementation. In this chapter we address the main questions of this dissertation: Can we construct more efficient implementations by using broadcast protocols that provide a weaker form of ordering? For which kinds of problems will this be successful?

We start by considering implementations based on a causal broadcast (CBCAST). We give a necessary and sufficient condition for a specification to be implementable with this type of broadcast. If such an implementation exists it can be expressed in a standard form. Finally, we show that a CBCAST implementation can be translated into an implementation based on FBCAST or even unordered broadcasts.

The implementations we construct this way can be characterized as follows: When a client invokes an operation, the return value can always be computed immediately from local information. This way the client need not wait for messages

to arrive at other sites or for replies to make it back; information is propagated asynchronously to other sites. Therefore we call this type of implementation *asynchronous*.

4.1 Causality and Timestamps

In Chapter 2 we introduced the idea of *Potential Causality* [Lam78]. In our execution model we can define this relation as follows.

Definition 4.1

The information flow relation "$\rightarrow$" on the events in an execution history E is the transitive, reflexive closure of "$\xrightarrow{D}$".

Two events a, b that are not related under "$\rightarrow$" are called concurrent ($a // b$).

If we interpret an execution history E as a directed graph (the nodes are the invocation and receive events in E; the edges are given by "$\xrightarrow{D}$") then $a \rightarrow b$ if and only if there is a path from a to b in this graph. Because "$\xrightarrow{D}$" is acyclic (Definition 3.7) the information flow relation "$\rightarrow$" defines a partial order on the events in an execution history.[1] The intuitive meaning of this relation is the following. An event a can affect some other event b only if it precedes b in this partial order. In particular, the state of a processor after an event b depends only on events that precede b under "$\rightarrow$". This fact is expressed in the next lemma.

[1] However, note that we define "$\rightarrow$" to be reflexive ($\forall e \in E: \quad e \rightarrow e$), contrary to the usual definition of a partial order. This notational convenience makes the later definitions simpler.

Definition 4.2

E' is a prefix of E ($E' \leq E$) iff

(i) $E' \subseteq E$

(ii) $\forall i\colon\ \forall a, b \in E'_i\colon\ a <_i b \text{ in } E' \ \Leftrightarrow\ a <_i b \text{ in } E$

(iii) $\forall a, b \in E\colon\ b \in E' \wedge a \rightarrow b \ \Rightarrow\ a \in E'$

For an event $a \in E$, we define $E[a]$ (the prefix at a) as follows:

(i) $E[a] \leq E$

(ii) $\forall$ invocation event $a' \in E\colon\ a' \in E[a] \ \Leftrightarrow\ a' \rightarrow a$

Lemma 4.1

Let E and E' be two execution histories and $a = E[i,j] = E'[i,j]$ be an event occurring in both histories ($a \in E \cap E'$).

If $E'[a] = E[a]$ then

(i) $stat_{E'}[i,j] = stat_E[i,j]$

(ii) $msg_{E'}\langle i,l\rangle = msg_E\langle i,l\rangle$ if $a = inv_E\langle i,l\rangle$ is an invocation event

$val_{E'}\langle i,l\rangle = val_E\langle i,l\rangle$ if $a = rcv_E(\langle i,l\rangle, i)$ is a local receive event

In other words, if we take an execution history E and modify it into a history E' in such a way that events preceding a under "$\rightarrow$" are unchanged (i.e., $E[a] = E'[a]$) then the state of processor i after event a as well as the message sent or the value returned to the client will not be affected by these modifications. Hence the lemma tells us that $E[a]$ contains exactly those events in E that have an effect on the outcome of a.

Proof: By induction on the number of events in $E[a]$. In the base case $E[a]$ contains only a single event, namely a. Then a must be the first event in E_i, i.e.,

$j = 1$; otherwise the event preceding a at i would also be in $E[a]$. Furthermore, a cannot be a receive event; otherwise the corresponding invocation event would precede a under "$\rightarrow$"and would be in $E[a]$. Therefore by Definition 3.10

$$stat_E[i,j] = stat_E[i,1] = \phi_i^s(stat_E[i,0], a) = \phi_i^s(q_0, a).$$

Because $stat_E[i,0] = stat_{E'}[i,0] = q_0$ we have $stat_{E'}[i,0] = stat_E[i,0]$. Furthermore by Definition 3.11

$$msg_E\langle i,1\rangle = \phi_i^m(stat_E[i,0], a) = \phi_i^m(q_0, a) = msg_{E'}\langle i,1\rangle.$$

For the induction step consider $E[a]$ with more than one event. Let $s = stat_E[i, j-1]$ and $s' = stat_{E'}[i, j-1]$. If $j = 1$ (a is the first event at p_i) then $s = s' = q_0$. Otherwise let $b = E[i, j-1]$ and $b' = E'[i, j-1]$ be the events preceding a at E_i and E'_i. Then $b \rightarrow a$ and $b' \rightarrow a$; hence $b \in E[a], b' \in E'[a]$. Because $E'[a] = E[a]$ we have $b' = b$ and $E'[b] = E[b] \leq E[a]$. By induction hypothesis the state of p_i after b is the same in E and E'; hence again $s = s'$.

If a is an invocation event then

$$stat_E[i,j] = \phi_i^s(s, a) = \phi_i^s(s', a) = stat_{E'}[i,j],$$
$$msg_E\langle i,j\rangle = \phi_i^m(s, a) = \phi_i^m(s', a) = msg_{E'}\langle i,j\rangle.$$

Otherwise $a = rcv_E(\langle j,l\rangle, i)$ is a receive event. Let $c = inv_E\langle j,l\rangle$ and $c' = inv_{E'}\langle j,l\rangle$ be the corresponding invocation events in E and E'. Then $c \rightarrow a$ and $c' \rightarrow a'$. It follows that $c, c' \in E[a] = E'[a]$, and therefore $c = c'$ and $E'[c] = E[c]$. By induction hypothesis $msg_E\langle j,l\rangle = msg_{E'}\langle j,l\rangle$. Therefore

$$stat_E[i,j] = \psi_i^s(s, msg_E\langle i,l\rangle) = \psi_i^s(s', msg_{E'}\langle i,l\rangle) = stat_{E'}[i,j],$$

and if $j = i$ (a local message was received) then

$$val_E\langle i,l\rangle = \psi_i^v(s, msg_E\langle i,l\rangle) = \psi_i^v(s', msg_{E'}\langle i,l\rangle) = val_{E'}\langle i,l\rangle.$$

□

Corollary 4.1

Let $E' \leq E$ be a prefix of the execution history E. Then

$\forall\, a = E'[i,j] \in E'$:

(i) $stat_{E'}[i,j] = stat_E[i,j]$

(ii) $msg_{E'}\langle i,l\rangle = msg_E\langle i,l\rangle$ if $a = inv_E\langle i,l\rangle$ is an invocation event

$val_{E'}\langle i,l\rangle = val_E\langle i,l\rangle$ if $a = rcv_E(\langle i,l\rangle, i)$ is a local receive event

Proof: $E' \leq E$ implies $E'[a] = E[a]$ for all $a \in E'$. □

In Section 3.2.2 we defined $H[E,i]$ to be the sequence of formal events observed by a particular client in an execution E of an implementation Y. The "$\rightarrow$" relation induces a partial order on on the formal events in the union of all $H[E,i]$. We call this partially ordered set of formal events derived from E and Y a *run*. It is defined formally below:

Definition 4.3

Given an execution history E and an implementation Y we define the run $R_Y(E)$ to be the set of formal events given by E, partially ordered by "$\rightarrow$":

$$R_Y(E) = \{event_E\langle i,j\rangle \mid \text{ for all } i,j\}$$

with the partial order "$\rightarrow$" on $R_Y(E)$ defined as

$$event_E\langle i,j\rangle \rightarrow event_E\langle l,m\rangle \;\Leftrightarrow\; inv_E\langle i,j\rangle \rightarrow inv_E\langle l,m\rangle$$

As in the case of an execution history we use the notation $event_R\langle i,j\rangle$ to denote the j'th event at processor i in a run R.

In [Lam78] Lamport introduced *logical timestamps*, integers assigned to each event in such a way that if all events are ordered by their timestamp this order is consistent with "$\rightarrow$". We can generalize this idea to timestamps which are vectors of integers [Sch85].[2] Such timestamps are useful for keeping track of the partial order of events as the system executes.

Definition 4.4

A timestamp t for an event $e = event_R\langle i,j\rangle \in R$ is a vector of n integers with the following meaning:

$$t_e[k] = \ \| \{event_R\langle k,l\rangle \in E_k \mid event_R\langle k,l\rangle \rightarrow e\} \|$$

i.e., $t_e[k]$ is the number of events at k that precede e in the partial order.

The following lemma states that given only the timestamp of two events in a run one can deduce their order under "$\rightarrow$".

Lemma 4.2

Let $a = event_R\langle i,j\rangle$ and $b = event_R\langle k,l\rangle$ be two events in a run R, and let t_a and t_b be their timestamps. Then

$$a \rightarrow b \ \Leftrightarrow \ t_a[i] \leq t_b[i]$$

Proof: $a = event_R\langle i,j\rangle$ is the j'th event invoked at site i. Therefore $event_R\langle i,j'\rangle \rightarrow a$ iff $j' \leq j$, and hence $t_a[i] = j$.

---

[2]The idea of vector timestamps was developed independently by Ladin and Liskov [LL86].

If $a \rightarrow b$ then by transitivity of "$\rightarrow$" $event_R\langle i, j' \rangle \rightarrow a$ for all $j' \leq j$. Hence $t_b[i] \geq j$.

Conversely, let $l = t_b[i] \geq j$. Then there are at least j events at processor i that precede b under "$\rightarrow$". In particular, there must be an event $event_R\langle i, j' \rangle \rightarrow b$ for some $j' \geq j$. which implies that $a \rightarrow event_R\langle i, j' \rangle$, and by transitivity $a \rightarrow b$. □

We will use these timestamps in the implementations we construct in the next section.

4.2 Cbcast Implementation

The *causal broadcast protocol* described by Birman and Joseph in [BJ87b] is a protocol that preserves the information flow relation between events, i.e., whenever two broadcasts b_1, b_2 are related under "$\rightarrow$" ($b_1 \rightarrow b_2$), the protocol guarantees that b_1 will be received before b_2 everywhere. Concurrent broadcasts may be received in different orders at different sites. In our formalism we define the ordering properties of CBCAST by the following axiom:

Definition 4.5

CBCAST ordering axiom:

(i) Causal ordering:

$$inv_E\langle i, j \rangle \rightarrow inv_E\langle l, m \rangle \;\Rightarrow\; \forall\, k\colon\; rcv_E(\langle i, j \rangle, k) <_k rcv_E(\langle l, m \rangle, k)$$

(ii) Immediate local delivery:

$$\forall\, i, j\colon\; \neg\, \exists\, a\colon\; inv_E\langle i, j \rangle <_k a <_k rcv_E(\langle i, j \rangle, i)$$

How can we use such a broadcast to construct an implementation for a given specification $\mathcal{S}$? Our plan is to take the ABCAST implementation from the previous chapter, replace the ABCAST by a CBCAST, and determine under which condition

the implementation will still be correct. In order for this to work it is necessary to make two more modifications to the ABCAST implementation:

- Recall that the correctness of the ABCAST implementation depended on the fact that all processors agreed on the order of events and therefore construct the same legal history H. If we use a CBCAST instead of ABCAST this will no longer be the case. However, using the timestamps introduced in the previous section it is possible for all processors to keep track of and agree upon the *partial* order of events during the execution of the system. In other words, we replace the the variable H in Figure 3.4 by a variable R, containing a partially ordered set of events (a run).

- Now, in order to execute an operation correctly it is necessary to relate these runs to globally ordered histories as they appear in a formal specification. For this purpose we introduce a function that maps partially ordered sets of events to totally ordered histories. We call this a *linearization operator*; it is defined formally below.

Definition 4.6

A linearization operator, $LIN : \mathcal{R} \rightarrow \mathcal{H} \cup \{\perp\}$, is a partial function[a] from runs to histories, such that:

(i) $LIN(\emptyset) = \emptyset$

(ii) $\forall R$: If $H = LIN(R) \neq \perp$ then

$$\forall a: \; a \in H \;\Leftrightarrow\; a \in R$$

$$\forall a, b \in R: \; a \rightarrow b \;\Rightarrow\; a <_H b$$

[a] The symbol $\perp$ denotes an undefined return value, i.e., $LIN(R) = \perp$ means LIN is undefined on R.

Processor i runs the following program:

R := empty;

$t := [0, 0, \ldots, 0]$;

$t[i] := 1$;

loop

wait for an invocation by the local client **or** the receipt of a broadcast;

if client invoked operation a **then**

pick a value v, such that $LIN(R + a{:}v) \in S$

CBCAST $(a{:}v, t)$ **to** all processors;

return v to the client;

else if broadcast $(a{:}v, t)$ was received **from** j **then**

$R := R \oplus_t a : v$;

t[j] := t[j] + 1;

end if

end loop

Figure 4.1: CBCAST implementation

For every run R on which $LIN(R)$ is defined, LIN linearizes the events in R in a way that preserves the partial order "$\rightarrow$" in R. With these changes, our CBCAST implementation, in informal pseudocode, looks like Figure 4.1.

We need to answer two questions.

1. For which kind of specifications will this CBCAST implementation be correct? We answer this question in Section 4.2.1.

2. How general is this implementation? Perhaps there are other methods of constructing a CBCAST implementation that cover a larger class of specifications. We address this question in Section 4.2.2.

In order to answer these questions we need to translate the CBCAST implementation from Figure 4.1 into our execution model.

Definition 4.7

We use the following notation:

$R + e$	the run R with the event e added at the end, i.e., ordered after *all* other events in R ($\forall e' \in R + e$: $e' \rightarrow e$).
$R' \leq R$	R' is a prefix of R $R' \subseteq R \ \wedge \ \forall e, e' \in R$: $(e \rightarrow e' \wedge e' \in R') \ \Rightarrow \ e \in R'$.
$R[e]$	The run consisting of all events preceding e in R, i.e., $R[e] = \{e' \in R \mid e' \rightarrow e\}$[3]
$R' \oplus_t e$	the run R with the event e added and ordered according to its timestamp t_e, i.e., $e \rightarrow event_R\langle i, k\rangle \ \Leftrightarrow \ t_e[i] \geq k$.

The implementation outlined in Figure 4.1 contains a construct

"pick a value v, such that $LIN(R + A_i{:}v) \in S$"

similar to the one use in Figure 3.4 (ABCAST implementation). In the ABCAST case we used the fact that specifications are complete to argue that such a return value will always exist. In the CBCAST implementation the argument no longer

[3]Recall that we defined "$\rightarrow$" to be reflexive. Therefore the event e is always contained in $R[e]$

holds, because, as defined in Definition 4.6, LIN may reorder events in $R \oplus A_i : v$ differently for different values of v. Therefore we must place the following restriction on LIN:

Definition 4.8

A linearization operator LIN is constructive under $S = (n, I, V, S)$ iff

$$\forall \text{ runs } R \text{ such that } LIN(R) \in S: \quad \forall \text{ invocations } a \in I:$$
$$\exists \text{ a return value } v \in V: \quad LIN(R + a{:}v) \in S.$$

If LIN is constructive under S we can define an *execution function* $\chi_{S,LIN} : S \times I \rightarrow V$ (similar to the one used in Section 3.5) with following property:

$$\forall R \text{ such that } LIN(R) \in S: \quad \forall a \in I:$$
$$v = \chi_{S,LIN}(R, a) \quad \Rightarrow \quad LIN(R + a{:}v) \in S$$

Given a specification $S = (n, I, V, S)$ and a constructive linearization operator LIN, we define the implementation $Y_{S,LIN}$ as follows:

$$Y_{S,LIN} = (n,\ I,\ V,\ I \times T,\ (I \times V)^{\#} \times T,\ (\emptyset, [0, \ldots, 1, \ldots, 0]),\ \Phi,\ \Psi),$$
$$\text{i.e., } M = I \times T,\ Q = (I \times V)^{\#} \times T,\ q_0 = (\emptyset, [0, \ldots, 1, \ldots, 0]),$$

where $A^{\#}$ denotes the set of all partially ordered subsets of A, i.e., $(I \times V)^{\#}$ is the set of all runs than can be constructed from events in $I \times V$. $T = N^n$ is the set of all timestamps (integer valued vectors of length n). The transition functions ϕ_i and ψ_i are defined as follows. When operation a is invoked at processor i in state $[R, t]$, it does not change its state but broadcasts $[a{:}v, t, i]$:

$$\phi_i([R, t], a) = ([R, t], [a{:}v, t, i]), \text{ where } v = \chi_{S,LIN}(R, a).$$

When processor i receives a message $[a{:}v, s, j]$, it adds the event $a{:}v$ to its run R, and updates its timestamp vector by incrementing the j'th component; the value v is returned to the client:

$$\psi_i([R,t],[a{:}v,s,j]) = ([R \oplus_e a{:}v, t'], v),$$
$$\text{where } t'[j] = t[j]+1, t'[k] = t[k] \text{ for } k \neq j.$$

4.2.1 Correctness of Cbcast Implementation

The implementation we defined above has the property that every run $R_Y(E)$ generated by one of its executions, satisfies a property that we call *local correctness*:

Definition 4.9

A run R is locally correct under LIN and S, iff

$$\forall \text{ events } e \in R: \quad LIN(R[e]) \in S$$

This property can be interpreted as follows. Say $e = a{:}v$ is an event invoked at processor i. In a run $R = R_{Y_{S,LIN}}(E)$ of a CBCAST execution, the subrun $R[e]$ contains exactly those events that processor i knows about at the time a is invoked. This is because $e' \in R[e]$ implies $e' \rightarrow e$; hence the CBCAST ordering guarantees that the message about e' is received before a is invoked. $LIN(R[e]) \in S$ then means that processor i executes the operation a in a way that is correct with respect to its local knowledge.

As stated in the next theorem, our CBCAST implementation $Y_{S,LIN}$ will be correct if the specification S has the property that local correctness always implies global correctness (i.e., $LIN(R) \in S$).

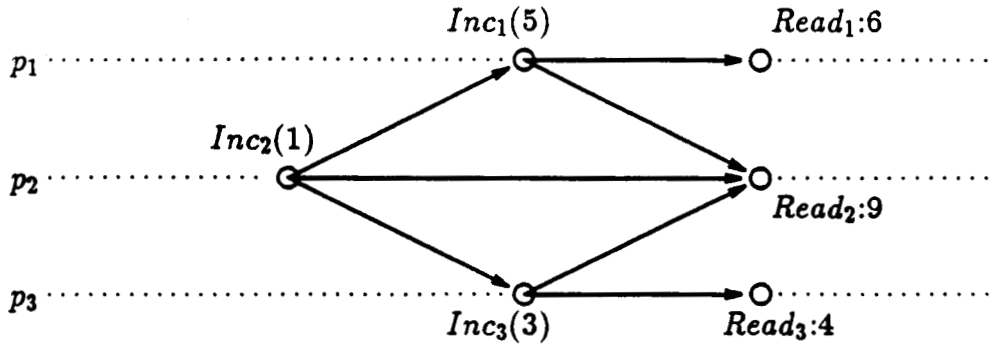


Figure 4.2: A locally correct run

Theorem 4.1

If LIN is constructive and satisfies

$$\forall \text{ runs } R\text{: } R \text{ locally correct } \Rightarrow LIN(R) \in S.$$

then $Y_{S,LIN}$ is a correct CBCAST implementation of specification S.

Before we present the proof it is useful to give an example for a specification that does not satisfy this condition. Consider the problem of implementing a simple counter. Clients can increment the counter by a specified amount (INC(X)) and read the current value of the counter (READ). It is straight forward to write down a specification for such a counter: in a legal history H every READ must return the sum of all increment values of INC operations preceding the READ in H. Figure 4.2 gives an example of a run R containing INC and READ events (events are represented by circles, the partial order by the arrows in the figure). This run satisfies local correctness. Consider for example

$$R[Read_2\text{:}9] = (Inc(1) \rightarrow Inc(5) // Inc(3) \rightarrow Read\text{:}9)$$

There are two ways of linearizing this subrun:

$$LIN(R[Read_2{:}9]) = (Inc_2(1), Inc_1(5), Inc_3(3), Read_2{:}9), \text{ or}$$
$$LIN(R[Read_2{:}9]) = (Inc_2(1), Inc_3(3), Inc_1(5), Read_2{:}9),$$

which are both legal histories. Similarly one can check that for every event e in this run, any linearization of $R[e]$ is legal. Therefore R is locally correct no matter how LIN is defined. But R itself can not be linearized into a legal history: If LIN orders $Inc_1(5)$ before $Inc_3(3)$ then the event $Read_3{:}4$ will be illegal; if $Inc_3(3)$ is ordered before $Inc_1(5)$ then $Read_1{:}6$ is illegal. Hence although R is locally correct it does not satisfy $LIN(R) \in S$ (global correctness).

We will give the proof for Theorem 4.1 on page 64 after the following three lemmas.

Lemma 4.3

(i) $E' \leq E \;\Rightarrow\; R_Y(E') \leq R_Y(E)$

(ii) Let $e = a{:}v \in R_Y(E)[a]$. Then
$R_Y(E)[e] = R_Y(E[a])$.

Proof: Follows from Definition 4.2 and Lemma 4.1. □

The next lemma makes a statement about the state of a processor in $Y_{S,LIN}$ at the time when a processor completes a client request and returns a value to the client.

Lemma 4.4

Let E be a CBCAST execution history and $R = R_{Y_{S,LIN}}(E)$. Let $e = a{:}v = event_E\langle i, j\rangle$ be an event in R and let $E[i, l] = rcv_E(\langle i, j\rangle, i)$ be the corresponding receive event in E. Then

$$stat_E[i, l] = [R[e], \text{timestamp}(e)].$$

In other words, at the time a value is returned to client i the state of p_i correctly reflects the timestamp of the event e as well as the run $R[e]$ of all events preceding e under "$\rightarrow$".

Proof: Let $stat_E[i,j] = [r,t]$.

(i) We will first show that r and $R[e]$ contain the same set of events. Let $e' = event_E\langle i',j'\rangle \in r$. Then e' was added to r when p_i received a message $m = [e',t',i']$ from processor i'. Therefore

$$rcv_E(\langle i',j'\rangle,i) <_i E[i,l] = rcv_E(\langle i,j\rangle,i)$$

Because of immediate local delivery (CBCAST axiom, Definition 4.5) this implies

$$rcv_E(\langle i',j'\rangle,i) <_i inv_E\langle i,j\rangle$$

Therefore $inv_E\langle i',j'\rangle \rightarrow inv_E\langle i,j\rangle$. By Definition 4.3 $e' \rightarrow e$ in R; hence $e' \in R[e]$. This shows $r \subseteq R[e]$.

Conversely consider $e' = event_E\langle i',j'\rangle \in R[e]$; then $e' \rightarrow e$ in R. By Definition 4.3 $inv_E\langle i',j'\rangle \rightarrow inv_E\langle i,j\rangle$. Because of causal ordering under the CBCAST axiom

$$rcv_E(\langle i',j'\rangle,i) <_i rcv_E(\langle i,j\rangle,i).$$

In other words, p_i receives the message $m = [e',t',i']$ from processor i' before $E[i,l] = rcv_E(\langle i,j\rangle,i)$. Hence the event e' will have been added to r by that time, i.e., $e' \in r$. This shows that $R[e] \subseteq r$. We conclude that (as unordered sets of events) $r = R[e]$.

(ii) Next, we show t = timestamp(e). The vector component $t[j]$ is incremented each time p_i receives a message from p_j. Therefore

$$t[j] = \parallel \{e' \in r \mid e' \text{ is an event invoked at } p_j\} \parallel$$

Part (i) of the proof implies

$$\{e' \in r \mid e' \text{ is an event invoked at } p_j\}$$
$$= \{e' \in R[e] \mid e' \text{ is an event invoked at } p_j\}$$

Therefore by Definition 4.4, $t = \text{timestamp}(e)$

(iii) Finally, we show that the partial orders in r and $R[e]$ are identical. This follows immediately from (i) and (ii), because in r events are ordered by timestamp. □

Lemma 4.5

If LIN is constructive and satisfies

$$\forall \text{ runs } R\text{: } R \text{ locally correct } \Rightarrow LIN(R) \in S.$$

then for every CBCAST execution history E: $R_{Y_{S,LIN}}(E)$ is locally correct.

Proof: Let $Y = Y_{S,LIN}$. We proceed by induction on the number of events in E. The base case, $E = \emptyset$, is trivially satisfied, because an empty run is always locally correct.

Induction step: consider $E \neq \emptyset$. We have to show that $LIN(R_Y(E)[e]) \in S$ for every e in $R_Y(E)$. Let $a = inv_E\langle i, j\rangle$ be an invocation event in E, and let $e = a{:}v = event_E\langle i, j\rangle$.

Define $E' = E[a]$, and $E'' = E' - a$. By Lemma 4.3 $R[e] = R_Y(E') = R_Y(E'') + a{:}v$. By induction hypothesis $R_Y(E'')$ is locally correct and therefore by assumption $LIN(R_Y(E'')) \in S$.

By Lemma 4.4 $R_Y(E'')$ is equal to the "R-part" of the state $stat_E[i, j]$ of processor i at the invocation event a. The message it sends is determined by ϕ:

$$m = \phi_i^m(R_Y(E''), a) = [a{:}v', t, i], \text{ where } v' = \chi_{S,LIN}(R_Y(E''), a).$$

Therefore $v = v' = \chi_{S,LIN}(R[e], a)$. From the definition of $\chi_{S,LIN}$ it follows that $LIN(R[e]) \cdot a:v) \in S$. □

We now have all the necessary tools to prove the theorem about the correctness of the CBCAST implementation.

Proof of Theorem 4.1: We show that under the assumption of Theorem 4.1 (local correctness of R implies $LIN(R) \in S$), for every CBCAST execution history E, the history $H = LIN(R_Y(E)) \in S$ and satisfies the correctness and liveness conditions of Definition 3.12.

(i) By Lemma 4.5 $R_Y(E)$ is locally correct and therefore by assumption

$$H = LIN(R_Y(E)) \in S.$$

(ii) **Correctness:** We have to show that $H|_i = H[E, i]$ for all i.
$H|_i$ contains the same set of events as $H[E, i]$, because $H = LIN(R_Y(E))$. Furthermore, the order $<_i$ on E_i is preserved in H, because $e <_i e'$ implies $e \rightarrow e'$, and LIN preserves "$\rightarrow$".

(iii) **Liveness:** Let $rcv_E(\langle l, m\rangle, k) <_k inv_E(l', m')$.
We have to show that $event_E\langle l, m\rangle < event_E\langle l', m'\rangle$ in H. This follows from the fact that LIN preserves "$\rightarrow$", because $rcv_E(\langle l, m\rangle, k) <_k inv_E(l', m')$ implies $event_E\langle l, m\rangle \rightarrow event_E\langle l', m'\rangle$. □

4.2.2 Existence of Cbcast Implementation

Theorem 4.1 in the previous section gave a sufficient condition for a specification S to be implementable with CBCAST. In this section we will show that this condition is not only sufficient but also *necessary*. This will show that our CBCAST implementation really is the most general implementation based on CBCAST: every problem

that has a CBCAST solution is solvable with our implementation. In other words, the goal of this section is to prove the following theorem.

Theorem 4.2

A specification S has a CBCAST implementation

$\Leftrightarrow$

$\exists$ constructive linearization operator LIN:

$\forall$ runs R: R locally correct $\Rightarrow$ $LIN(R) \in S$.

The only if ("$\Leftarrow$") direction is equivalent to Theorem 4.1 which we proved in the previous section. So, our task is the following: Given some CBCAST implementation Y of S, we have to derive a linearization operator that satisfies the conditions of Theorem 4.2. We will do this as follows: Given a run R our linearization operator will map this run to a history in two steps:

$$R \rightarrow E \rightarrow H$$

In the first step, we map a run R to an execution history E that has the same set of invocations as R and the same partial order as R. The second mapping is defined in terms of the behavior of implementation Y. If Y is correct then there must exist a legal history H that satisfies the correctness and liveness conditions with respect to E (Definition 3.12). We map E to this history.

The first mapping is called Γ. We want to define this mapping in such a way that it preserves the partial order "$\rightarrow$" on R. Given R we get a partially ordered set of invocation events simply by ignoring the return values of the formal events in R. The execution history $E = \Gamma(R)$ will have exactly these invocation events plus all the corresponding receive events. Notice that in a CBCAST execution history

the "$\rightarrow$" relation between invocation events already determines the order of receive events relative to invocation events in each processor history E_i:

$$rcv_E(\langle k,l\rangle, i) <_i inv_E\langle i,j\rangle \quad \Leftrightarrow \quad inv_E\langle k,l\rangle \rightarrow inv_E\langle i,j\rangle$$

The "$\Rightarrow$" direction follows from the definition of "$\rightarrow$", the other direction from the CBCAST ordering axiom (Definition 4.5). Therefore, to completely determine $E = \Gamma(R)$ we only need to specify the order of the receive events between two invocation events. The CBCAST ordering axiom already determines a partial order on these events; to define $E = \Gamma(R)$ we can pick any linearization of this partial order. A topological sort will suffice. The next definition summarizes this procedure.

Definition 4.10

The function $\Gamma : \mathcal{R} \rightarrow \mathcal{E}$ maps a run R to an execution history E in the following way:

(i) $E_i = \{a \mid \exists\, j, v\colon\ a{:}v = event_R\langle i,j\rangle \in R\}$
$\quad\quad \cup\ \{(k,l) \mid \exists\, k,l\colon\ event_R\langle k,l\rangle \in R\}$

(ii) $<_i$ is the topological sort of the partial order "$\rightarrow$" on the events in E_i induced by "$\rightarrow$" on R.

The next lemma formally states the properties of this mapping.

Lemma 4.6

Let $E = \Gamma(R)$. Then

(i) E is a CBCAST execution history.

(ii) $inv_E\langle i,j\rangle \rightarrow inv_E\langle i',j'\rangle$ in $E \quad \Leftrightarrow \quad event_R\langle i,j\rangle \rightarrow event_R\langle i',j'\rangle$ in R.

(iii) $R' \leq R \ \Rightarrow\ \Gamma(R') \leq \Gamma(R)$.

Proof: (i) and (ii): As discussed above, we constructed Γ in such a way as to satisfy these two properties.

(iii) Follows from property (i) and Definition 4.10(ii). □

We now use Γ to define a linearization operator LIN derived from an implementation of S.

Definition 4.11

Let $Y = (n, I, V, M, Q, q_0, \Phi, \Psi)$ be a CBCAST implementation of S.

$$\mathcal{H}(R) = \{H \in S \mid H \text{ is a linearization of } R_Y(\Gamma(R))\}$$

$$LIN_Y(R) = \begin{cases} \text{"smallest" } H \text{ in } \mathcal{H}(R) & \text{if } R = R_Y(\Gamma(R)) \\ \perp & \text{otherwise} \end{cases}$$

Notice that this definition implicitly assumes that $\mathcal{H}(R)$ is non-empty whenever $R = R_Y(\Gamma(R))$.

Lemma 4.7

If Y is a correct implementation of S then $R = R_Y(\Gamma(R))$ implies $\mathcal{H}(R) \neq \emptyset$, hence LIN_Y is well defined.

Proof: Let $E = \Gamma(R)$. If Y is correct then there exists a history $H \in S$ that satisfies the correctness and liveness condition of Definition 3.12. Correctness and liveness imply that H is a linearization of $R_Y(E)$. Therefore $H \in \mathcal{H}(R)$ if $R = R_Y(E)$. □

We now proceed to show that if Y is a correct CBCAST implementation of S then LIN_Y indeed satisfies the conditions of Theorem 4.2.

Lemma 4.8

LIN_Y is constructive.

Proof: Let R be a run such that $LIN_Y(R) \in S$. We have to show that for every invocation $a \in I$ there exists a return value v such that $LIN_Y(R + a{:}v) \in S$.

Let $E = \Gamma(R)$. If $LIN_Y(R) \neq \perp$, Definition 4.11 implies $R = R_Y(E)$. Consider an execution E' that is identical to E, except that it has one more invocation event a at the end, i.e.,

$$E'_i = E_i + a + (i, j+1) \quad \text{where } j \text{ is the number of invocation events in } E_i$$
$$E'_k = E_k + (i, j+1) \qquad \text{for } k \neq i$$

Let $R' = R_Y(E')$. Then $R' = R + a{:}v$, where $v = val_{E'}\langle i, j+1\rangle$. By construction $E' = \Gamma(R')$. Therefore, by Definition 4.11 $LIN_Y(R + a{:}v) = LIN_Y(R') \in S$. □

Lemma 4.9

$$\forall \text{runs } R\text{: } R \text{ locally correct } \Rightarrow \; LIN_Y(R) \in S.$$

Proof: Induction on the number of events in R. The base case, $R = \emptyset$, is trivially satisfied, because an empty history is always legal.

For the induction step consider $R \neq \emptyset$. We have to show that $LIN_Y(R) \in S$, which, by Definition 4.11, is equivalent to $R = R_Y(E)$, where $E = \Gamma(R)$. Let $a{:}v = event_R\langle i, j\rangle \in R$ be an event in R which corresponds to the invocation event $a = inv_E\langle i, j\rangle$ in E. To prove that $R = R_Y(E)$ we have to show that $val_E\langle i, j\rangle = v$.

Let e be a maximal event in R, and define $R' = R[e]$ and $R'' = R - \{e\}$. Local correctness of R implies that $LIN_Y(R') \in S$ and that R'' is locally correct. By induction hypothesis $LIN_Y(R'') \in S$. Therefore, by Definition 4.11, $R' = R_Y(E')$ and $R'' = R_Y(E'')$, where $E' = \Gamma(R')$ and $E'' = \Gamma(R'')$.

Case 1 $a{:}v = e$: In this case $a{:}v \in R'$. Because $R' \leq R$ we have $E' \leq E$ (Lemma 4.6). By Lemma 4.1, $val_E\langle i, j\rangle = val_{E''}\langle i, j\rangle = v$.

Case 2 $a{:}v \neq e$: In this case $a{:}v \in R''$. Because $R'' \leq R$ we have $E'' \leq E$ (Lemma 4.6). Again, by Lemma 4.1 $val_E\langle i,j\rangle = val_{E''}\langle i,j\rangle = v$. □

Proof of theorem 4.2: The "⇐" direction is equivalent to Theorem 4.1. The "⇒" direction follows from Lemma 4.8 and Lemma 4.9. □

4.3 Bcast and Fbcast Implementation

In this section we consider the problem of constructing implementations based on unordered or FIFO broadcasts (BCAST and FBCAST). We start by investigating BCAST implementations.

The CBCAST protocol presented in in [BJ87b] implements causal ordering on top of unordered message channels by a method called "piggybacking". Every broadcast message is augmented by previous messages it might depend on before the message is sent out. This way causal ordering can be achieved without multiple phases of message exchanges. We use this idea to translate our CBCAST implementation from Figure 4.1 into an equivalent BCAST implementation of the same specification. As in the CBCAST implementation every processor keeps track of all events and their partial order. When client i invokes an operation a, processor i not only broadcasts the event $e = a{:}v$, but also the whole set of events that precede e under "$\rightarrow$". An informal description of the BCAST implementation is given in Figure 4.3. In the rest of this section we will translate this implementation into our formal execution model and show that it is correct under exactly the same conditions for S and LIN as the CBCAST implementation.

Processor i runs the following program:

$R :=$ empty;

loop

wait for an invocation by the local client or the receipt of a broadcast;

if client invoked operation a then

pick a value v, such that $LIN(R + a{:}v) \in S$

$R := R + a{:}v$;

BCAST (R) **to all processors;**

return v to the client;

else if broadcast (R') was received then

$R := R \cup R'$;

end if

end loop

Figure 4.3: BCAST implementation

Given a specification $S = (n, I, V, S)$ and a constructive linearization operator LIN, we define the implementation $Y_{S,LIN}$ as follows:

$$Y_{S,LIN} = (n,\ I,\ V,\ (I \times V)^{\#} \times V,\ (I \times V)^{\#},\ \emptyset,\ \Phi,\ \Psi),$$
$$\text{i.e., } M = (I \times V)^{\#} \times V,\ Q = (I \times V)^{\#},\ q_0 = \emptyset,$$

where $(I \times V)^{\#}$ denotes the set of all runs than can be constructed from events in $I \times V$. The transition functions ϕ_i and ψ_i are defined as follows. When operation a is invoked at processor i in state R, it adds the event $a{:}v$ to its run R, and broadcasts $[R + a{:}v, v]$:

$$\phi_i(R, a) = (R + a{:}v, [R + a{:}v, v]),\ \text{where } v = \chi_{S,LIN}(R, a)$$

When processor i receives a message $[R', v]$, it adds all events in R' to its run R, and the value v is returned to the client:

$$\psi_i(R, [R', v]) = (R \cup R', v)$$

The next lemma makes a statement about the state of a processor in $Y_{S,LIN}$ at time when a processor completes a client request (i.e., returns a value to the client).

Lemma 4.10

Let E be a BCAST *execution history and $R = R_{Y_{S,LIN}}(E)$. Let $e = a{:}v = event_E\langle i, j\rangle$ be an event in R and let $E[i, l] = inv_E\langle i, j\rangle$ be the corresponding invocation event in E. Then*

$$stat_E[i, l] = R[e]$$

Proof: Let $stat_E[i, j] = r$. Proof by induction on the number of events in $R[e]$. Base case: $R[e] = \{e\}$. Then $a = inv_E\langle i, j\rangle$ is the first invocation event in E_i.

Furthermore, there can be no receive events preceding a in E_i; otherwise the corresponding formal events would be in $R[e]$. Therefore the state of p_i before the event e is $q_0 = \emptyset$. Hence $r = \emptyset + e = \{e\}$.

Induction step: consider r with more than one event. Let $f \in r$. Then e' was added to r when p_i received a message $m = r'$ with $f \in r'$ from processor i'. Let $e' = event_E\langle i', j'\rangle$ be the invocation that caused $p_{i'}$ to send this message. Then $e' \rightarrow e$, because $rcv_E(\langle i', j'\rangle, i) <_i inv_E\langle i, j\rangle$. Further more, by induction hypothesis $r' = R[e']$; hence $f \rightarrow e'$. By transitivity $f \rightarrow e$. Therefore $f \in R[e]$. This shows $R[e] \subseteq r$.

Let $e' = event_E\langle i', j'\rangle \in R[e]$, i.e., $e' \rightarrow e$. Then $inv_E\langle i', j'\rangle \rightarrow inv_E\langle i, j\rangle$. If $i' = i$ then e' was added to r at the invocation $inv_E\langle i, j'\rangle$; hence $e' \in r$. Otherwise, by definition of "$\rightarrow$"(4.1) there must be a receive event $rcv_E(\langle i'', j''\rangle, i) \in E_i$ such that

$$inv_E\langle i', j'\rangle \rightarrow inv_E\langle i'', j''\rangle \xrightarrow{D} rcv_E(\langle i'', j''\rangle, i) <_i inv_E\langle i, j\rangle.$$

Then $e' \rightarrow e'' = event_E\langle i'', j''\rangle$. By induction hypothesis the state r'' of $p_{j''}$ after the invocation event $inv_E\langle i'', j''\rangle$ is equal to $R[e'']$. Therefore $e' \in r'' = msg_E\langle i'', j''\rangle$. Therefore, when p_i receives the message from $p_{j''}$ it contains e' which will then be added to r. This shows $r \subseteq R[e]$. We conclude that $r = R[e]$. □

This lemma is the equivalent of Lemma 4.4 for CBCAST implementations.

Theorem 4.3

If LIN is constructive and satisfies

$$\forall \text{ runs } R\text{: } R \text{ locally correct } \Rightarrow LIN(R) \in S.$$

then $Y_{S,LIN}$ is a correct BCAST implementation of specification S.

Proof: The proof is the same as for Theorem 4.1 with references to Lemma 4.4 replaced by Lemma 4.10 □

Theorem 4.4

A specification S has a BCAST implementation

$\Leftrightarrow$

$\exists$ constructive linearization operator LIN:

$\forall$ runs R: R locally correct $\Rightarrow$ $LIN(R) \in S$.

Proof: The "$\Leftarrow$" direction is equivalent to the previous theorem (4.3). The "$\Rightarrow$" direction follows from theorem 4.2, because every BCAST implementation for S is also a CBCAST implementation. □

Corollary 4.2

A specification S has a FBCAST implementation

$\Leftrightarrow$

$\exists$ constructive linearization operator LIN:

$\forall$ runs R: R locally correct $\Rightarrow$ $LIN(R) \in S$.

Proof: The "$\Leftarrow$" direction follows from theorem 4.4, because every BCAST implementation for S is also a FBCAST implementation. The "$\Rightarrow$" direction follows from theorem 4.2, because every FBCAST implementation for S is also a CBCAST implementation. □

4.4 Summary

In this chapter we looked at the problem of constructing an implementation for a formal specification using broadcast protocols that are more efficient than ABCAST. Let $\mathcal{S}$ be the class of all formal specification and let $\mathcal{S}_{xbcast}$ be the the subset of $\mathcal{S}$ containing all specifications that have an XBCAST implementation (where XBCAST stands for ABCAST, CBCAST, ...). We have shown that $\mathcal{S}$ separates into two distinct subclasses:

$$\mathcal{S} = \mathcal{S}_{abcast} \;>\; \mathcal{S}_{cbcast} = \mathcal{S}_{fbcast} = \mathcal{S}_{bcast}.$$

We call the second class $\mathcal{S}_{async}$ because specifications in this class have implementations with the following characteristic. When a client invokes an operation it is always possible to compute the return value immediately from local information. This way the client never has to wait for replies from remote sites; information is propagated asynchronously in the background.

We showed that a specification S is a member of the class $\mathcal{S}_{async}$ if and only if there exists a linearization operator for S (Theorem 4.2). This linearization operator can be used to automatically construct an implementation for S. In the next chapter we will look at the problem of finding such an operator.

Chapter 5

Commutative Specifications

In the previous section we gave a complete characterization for the class of specifications that have an asynchronous implementation. Unfortunately as we will show in the next section, this class is non-recursive, i.e., in general the question of whether a specification has an asynchronous implementation is undecidable. This result shows that we cannot find a general algorithm that would automatically construct a suitable linearization operator from a given specification. Therefore, we have to investigate methods that could be applied to certain "simple" subclasses of specifications. In this chapter we explore the possibility of exploiting knowledge about the commutativity of operations in a specification in order to construct linearization operators.

5.1 Undecidability

Theorem 4.2 reduces the problem of constructing an asynchronous implementation for a specification S to the problem of finding a linearization operator LIN that satisfies the condition of Theorem 4.2. Unfortunately this problem is still very hard.

In fact, the example below shows that the general problem of deciding whether a specification S has an asynchronous implementation is undecidable.

Consider a system with two processors in which client 1 may invoke a parameterless operation a; client 2 may invoke an operation b with one integer parameter. Define the following class of specifications:

$S_i = (2, I, V, S_i)$, where

$$
\begin{aligned}
I &= \{a_1\} \cup \{b_2(x) \mid x \in N\} \\
V &= \{0,1\} \\
S_i &= \{\ (a_1{:}0)\ \} \\
&\cup\ \{\ (b_2(x){:}0)\ \mid x \in N\} \\
&\cup\ \{\ (a_1{:}0, b_2(x){:}0)\ \mid x \notin h_i\} \\
&\cup\ \{\ (a_1{:}0, b_2(x){:}1)\ \mid x \in h_i\} \\
&\cup\ \{\ (b_2(x){:}0, a_1{:}1)\ \mid x \in N\}
\end{aligned}
$$

where

$$
\begin{aligned}
h_i &= \{x \mid x \text{ is an encoding of a computation of the} \\
&\qquad i\text{'th Turing machine, } T_i\text{, in which } T_i \text{ halts}\}
\end{aligned}
$$

Lemma 5.1

S_i has an asynchronous implementation iff the Turing machine T_i never halts.

Proof: Let LIN be a linearization operator that satisfies the condition of Theorem 4.2 for S_i. Consider the run

$$R = a_1{:}0 \ //\ b_2(x){:}0$$

Table 5.1: Linearization operator for S_i

R	$LIN(R)$	
a_1:0	$(a_1$:0)	
a_1:1	$\perp$	
$b_2(x)$:0	$(b_2$:0)	for all $x \in N$
$b_2(x)$:1	$\perp$	for all $x \in N$
a_1:0 // $b_2(x)$:0	$(a_1$:0, $b_2(x)$:0)	for all $x \in N$
a_1:u // $b_2(x)$:v	$\perp$	if $u \neq 0$ or $v \neq 0$
a_1:0 $\rightarrow$ $b_2(x)$:0	$(a_1$:0, $b_2(x)$:0)	for all $x \in N$
a_1:u $\rightarrow$ $b_2(x)$:v	$\perp$	if $u \neq 0$ or $v \neq 0$
$b_2(x)$:0 $\rightarrow$ a_1:1	$(b_2(x)$:0, a_1:1)	for all $x \in N$
$b_2(x)$:u $\rightarrow$ a_1:v	$\perp$	if $u \neq 0$ or $v \neq 1$
all other R	$\perp$	

for some $x \in N$. If LIN is constructive then $LIN(\emptyset) = \emptyset \in S_i$ implies that there exists a return value v such that $LIN(a{:}v) \in S_i$. The way S_i is defined this is only possible if $v = 0$. Hence $LIN(a_1{:}0) = (a_1{:}0) \in S_i$, and by the same argument $LIN(b_2(x){:}0) = (b_2(x){:}0) \in S_i$. Therefore the run R satisfies *local correctness*. By Theorem 4.2 $LIN(R) \in S_i$. This is only possible if $LIN(R) = (a_1{:}0, b_2(x){:}0)$, and $x \notin h_i$. But R was locally correct for any x. Hence $h_i = \emptyset$, i.e., T_i never halts.

Conversely, assume that T_i never halts. Define a linearization operator LIN by Table 5.1. The way S_i is defined, a legal history has at most one event at p_1 and one event at p_2. Consequently a locally correct run can have only two events. Therefore our table enumerates all possible locally correct runs. It is straight forward to check

that if $h_i = \emptyset$ then for every row in the table, either the history in the right column is legal, or the run in the left column violates local correctness. □

The lemma shows that, if we had a procedure for deciding if S_i is simple for a given i, then we would have solved the halting problem for Turing machines. But since the halting problem is undecidable we have:

Corollary 5.1

The problem of finding those i for which S_i is simple is undecidable.

Fortunately, hardly any problem that arises in real distributed systems has anything to do with Turing machine computations. In many cases the problem at hand can be solved despite the undecidability of the general case.

5.2 Commutative Specifications

The difference between an ABCAST execution history and a CBCAST execution history, is that in the CBCAST case different processors may observe events in different orders. Therefore, it should be easier to construct a CBCAST implementation if certain events commute, that is, if their order in a legal history can be reversed without making the history illegal. We explore this idea in this section.

5.2.1 Commutativity and Ordering Constraints

We start by defining an equivalence relation on histories. Two histories are equivalent if no sequence of future events can distinguish them.

Definition 5.1

Two histories, H and H_2 are equivalent ($H_1 \equiv H_2$) iff

$$\forall H:\ H_1 H \in S \ \Leftrightarrow\ H_2 H \in S.$$

We can identify the equivalence classes of histories with the states the system can be in. Different histories in the same equivalence class represent different ways of reaching the same system state. We use this equivalence relation to distinguish between *read-only* events and *update* events. An event is a read-only event if it does not change the system state.

Definition 5.2

An event e is read-only iff

$$\forall H:\ He \in S \ \Rightarrow\ H \equiv He.$$

Events that are not read-only are called update events.

Note that whether a particular operation is read-only depends on the outcome (i.e., return value) of the operation. Consider, for instance, the PASS operation from our token passing example. The event $P_i(x)$:*ok* is an update whereas $P_i(x)$:eH is a read-only event.

We now turn our attention to specifications in which update events always commute.

Definition 5.3

Specification S is commutative iff

$\forall H:\ \forall a, b$ update events at different processors:

$$Ha \in S \wedge Hb \in S \ \Rightarrow\ Hab \in S \wedge Hba \in S \wedge Hab \equiv Hba.$$

Clearly, read-only events always commute with each other, but read-only events may or may not commute with update events. Consider $Ha, Hb \in S$, where a is read-only and b is an update event. Then $Hab \in S$, otherwise a would not be read-only. It could be that also $Hba \in S$; this would mean that the return value in a is not affected by the update b. Otherwise, if $Hba \notin S$, then a is affected by b, i.e., the return value in a is no longer valid if a is ordered after b. We denote this kind of dependency between two events by the symbol "$\mapsto$".

Definition 5.4

a is <u>invalidated</u> by b ($a \mapsto b$) iff

$$\exists H: \; Ha \in S \wedge Hb \in S \wedge Hba \notin S.$$

If $e_1 \mapsto e_2$ or $e_2 \mapsto e_1$ we also say that there is an *ordering constraint* between the two events. From the above discussion it is clear that

Lemma 5.2

If S is commutative then

$$a \mapsto b \;\Rightarrow\; a \text{ is read-only and } b \text{ is an update event.}$$

As an example let us consider various different events possible in our token passing service. The read-only events are

$$Q_i{:}T,\ Q_i{:}F,\ R_i{:}eH,\ R_i{:}eR,\ P_i(j){:}eH,\ P_i(j) : eR,$$

whereas the following are update events:

$$R_i{:}ok,\ P_i(j){:}ok.$$

Note that in the traditional sense, PASS and REQUEST operations do not commute.

For example, the history

$$H = \langle\, R_3{:}ok,\ P_1(3){:}ok \,\rangle$$

would not be legal if we reversed the order of the two events. If the PASS operation is invoked before the REQUEST operation it should return ERRORREQUEST instead of OK. However, according to our definition the token passing specification *is* commutative. This is because we require two update events a, b to commute only if both events are legal independent of each other ($Ha \in S$ and $Hb \in S$ for some H). Hence the fact that the two events $R_3{:}ok$ and $P_1(3){:}ok$ do not commute does not affect the commutativity of the specification, because the second event ($P_1(3){:}ok$) is never legal without the first. Formally:

$$\neg\, \exists\, H \in S\colon\ \ H + R_3{:}ok \in S\ \wedge\ H + P_1(3){:}ok \in S$$

A complete analysis of the token specification shows that any two update events either commute or have the property that one is never legal without the other (see Appendix A). Hence the token passing specification is commutative.

The intuitive reason for defining commutative specifications this way is the following. If there are two updates a, b of this type (b is not legal without a) then these two events will not occur concurrently in an execution. The two events will always be related by information flow ($a \rightarrow b$). Because CBCAST preserves "$\rightarrow$", all processors will observe a before b. Therefore it does not matter whether the two events commute.

In the token passing specification there are two types of ordering constraints: the first one between certain QUERY and PASS events

$$Q_i{:}F \ \mapsto\ P_j(i){:}ok,$$

the second one between an unsuccessful PASS event and a request event:

$$P_i(j){:}eR \vdash R_j{:}ok$$

A complete table of dependencies for token passing events is given in the appendix.

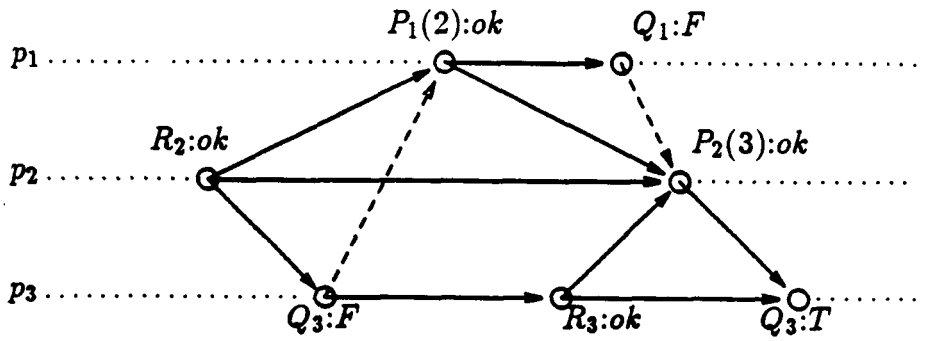
5.2.2 Applying Commutativity to Runs

How do the concepts discussed in the previous section help us construct a linearization operator for a commutative specification? Our plan is the following:

1. We assume that we can compute the ordering constraints "$\mapsto$" between any two pairs of events. Given a run R, we construct what we call the *closure* of R ($\overline{R}$) by adding extra edges to R: For all events $a, b \in R$ that are concurrent in R, we add an edge $a \rightarrow b$ if the ordering constraint $a \mapsto b$ holds.

2. Provided that $\overline{R}$ has no cycles, we define $LIN(R)$ by arbitrarily picking a linearization H of $\overline{R}$. That is, we pick a history H that contains the same events as R and has a total order that is consistent with "$\rightarrow$"and "$\mapsto$".

Figure 5.1 shows an example of applying this method to a run of the token passing service. It shows a run R and its closure. R is represented by the circles (events) and solid arrows (information flow relation between those events). $\overline{R}$ is given by R plus the dashed arrows (ordering constraints). We can get a legal history for this run by ordering all its events in such a way that the partial order given by the solid and dashed arrows is preserved. The history H given below the diagram shows one possible linearization. We formalize this method below and show that the linearization operator defined this way works in the case of commutative specifications.



$H = (\ R_2{:}ok,\ Q_3{:}F,\ P_1(2){:}ok,\ R_3{:}ok,\ Q_1{:}F,\ P_2(3){:}ok,\ Q_3{:}T\ )$

Figure 5.1: An example run and one of its linearizations

Definition 5.5

The closure $\overline{R}$ of a run R is the run R augmented by edges between any two concurrent events a and b, whenever $a \mapsto b$, or formally:

$$a \rightarrow b \in \overline{R} \ \Leftrightarrow\ (a \rightarrow b \in R \ \vee\ a//b \in R \wedge a \mapsto b)$$

Definition 5.6

$$\overline{\mathcal{H}}(R) = \{H \in S \mid H \text{ is a linearization of } \overline{R}\}$$

$$LIN_S(R) = \begin{cases} \text{"smallest" } H \text{ in } \overline{\mathcal{H}}(R) & \text{if } \overline{\mathcal{H}}(R) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

Recall from Theorem 4.1 that, to show that the CBCAST implementation will be correct with this linearization operator, we have to prove

$$LIN_S(R) \in S \quad \text{for every locally correct run } R,$$

where local correctness means $LIN(R[a]) \in S$ for every $a \in R$. As defined above, LIN_S, simply picks one possible linearization of $\overline{R}$ to map a run R to a history.

Hence in this case local correctness of R implies that every $\overline{R[a]}$ (for a in R) has a legal linearization. We call such a run *weakly plausible*:

Definition 5.7

R is weakly plausible $\Leftrightarrow$ $\forall a \in R$: $\exists$ legal linearization of $\overline{R[a]}$.

If not just one, but all linerizations of $\overline{R[a]}$ are legal, then we call this run *strongly plausible*:

Definition 5.8

R is strongly plausible iff

$\forall a \in R$: $\exists$ legal linearization of $\overline{R[a]}$

$\wedge$ every linearization of $\overline{R[a]}$ is legal.

The relationship between local correctness under LIN_S and strong and weak plausibility is the following: Strong plausibility implies local correctness, and local correctness implies weak plausibility. We will show (Lemma 5.4 below) that for commutative specifications these two forms of plausibility are in fact equivalent. Hence a run is locally correct if and only if it is plausible (strong or weak). Therefore we only need to show that $LIN_S(R) \in S$ for strongly plausible runs R. The next lemma will allow us to do this.

Lemma 5.3

If R is strongly plausible then every linearization of $\overline{R}$ is legal.

Proof: Induction on the number of events in R: Trivially satisfied for empty runs, because empty histories are always legal. Now assume R non empty:

Case 1: If R has a unique maximal element a then $R = R[a]$, and our claim follows from Definition 5.8.

Case 2: Let H be an arbitrary linearization of $\overline{R}$, let a be the last event in H, and let $b \neq a$ be a maximal element of R. H can be written as $H = H'bH''a$. The history $H_1 = H'bH''$ is a linearization of $\overline{R - a}$. By induction hypothesis H_1 is legal. Similar, $H_2 = H'H''a$ is legal as a linearization of $\overline{R - b}$. Let H''' be equal to H'', except that all read-only events are removed from H'''. If b is a read-only event then

$$H = H'bH''a \equiv H'H''a = H_2 \in S$$

and we are done. Otherwise, b commutes with every event in H'''; hence

$$H'H'''b \equiv H'bH''' \equiv H'bH'' = H_1 \in S, \text{ and}$$
$$H'H'''a \equiv H'H''a = H_2 \in S.$$

Then $H'H'''ba \in S$, because otherwise there would be an ordering constraint $a \mapsto b$, but then a could not be the last event in a linearization of $\overline{R}$. Therefore

$$H = H'bH''a \equiv H'bH'''a \equiv H'H'''ba \in S.$$

□

We can now prove that for commutative specifications weak and strong plausibility are equivalent.

Lemma 5.4

If S is commutative then

(i) Weak and strong plausibility are equivalent.

(ii) Every linearization of a plausible run is equivalent.

Proof: (i) We have to show that every weakly plausible run R is also strongly plausible. Induction on the number of events in R: Trivially satisfied for empty runs, because empty histories are always legal.

Now consider a non-empty, weakly plausible run R. Assume R is not strongly plausible. Then there must be a left subrun $\overline{R[a]}$ with a legal linearization, say Ha, such that some other linearization $H'a$ is not legal. These two histories only differ in the order of events that are concurrent in $\overline{R[a]}$. We may transform one into the other by swapping adjacent concurrent events. Thus we get a sequence of histories

$$H_1a,\ H_2a,\ H_3a,\ \ldots,\ H_na, \text{ where } H_1 = H \text{ and } H_n = H',$$

in which H_i and H_{i+1} differ only in the order of two adjacent events. If $H'a \notin S$ then the sequence must contain two consecutive histories,

$$H_i = Ab_1b_2B, \quad H_{i+1} = Ab_2b_1B$$

such that H_ia is legal but $H_{i+1}a$ is not. Note that H_i and H_{i+1} are linearizations of $R' = \overline{R[a]} - a$. By induction hypothesis R' is strongly plausible. By Lemma 5.3 H_ia and $H_{i+1}a$ are both legal. Because specifications are prefix-closed, Ab_1b_2 and Ab_2b_1 must also be legal. If S is commutative then these last two histories are equivalent, and hence H_ia and $H_{i+1}a$ should either both be legal or both be illegal. This contradicts our earlier assumption.

(ii) Let H and H' be two linearizations of a plausible run R. We have to show that $H' \equiv H$. H and H' differ in the order of events that are concurrent in $\overline{R}$. Again, we transform one into the other by swapping adjacent concurrent events, leading to a sequence of histories:

$$H_1,\ H_2,\ H_3,\ \ldots,\ H_n, \text{ where } H_1 = H \text{ and } H_n = H'.$$

C-2

We can write H_i and H_{i+1} as

$$H_i = Ab_1b_2B, \quad H_{i+1} = Ab_2b_1B.$$

From part (i) we know that R is strongly plausible; hence, by Lemma 5.3, H_i and H_{i+1} are both legal. Therefore their prefixes Ab_1 and Ab_2 are legal. If one of the two events (say b_1) is a read-only event then

$$Ab_1b_2 \equiv Ab_2 \equiv Ab_2b_1.$$

If both events are updates then they must commute. In any case, we have $Ab_1b_2 \equiv Ab_2b_1$ and therefore $H_i \equiv H_{i+1}$. By transitivity $H \equiv H'$. □

This lemma now allows us to show that the linearization operator we introduced in this section (Definition 5.6) can be used to construct asynchronous implementations.

Definition 5.9

A commutative specification S is acyclic iff

$$\forall R\text{: } R \text{ plausible} \Rightarrow \overline{R} \text{ acyclic.}$$

Otherwise we say S is cyclic.

Theorem 5.1

If S is commutative and acyclic then the CBCAST implementation with LIN_S as its linearization operator is correct.

Proof: We will show that if every plausible run has an acyclic closure then LIN_S is constructive and satisfies $LIN_S(R) \in S$ for every locally correct run R. The claim then follows from Theorem 4.2.

(i) LIN_S is constructive: Let $R \in S, a \in I$. We have to show that there is a return value $v \in V$ such that $LIN_S(R + a{:}v) \in S$. LIN_S is defined in such a way that the order of events in $H = LIN_S(R + a{:}v)$ is independent of the choice for the return value v, that is

$$LIN_S(R + a{:}v) = LIN_S(R) + a{:}v, \text{ for all } v \text{ such that } LIN_S(R) + a{:}v \in S$$

Because specifications are complete (Definition 3.3) $LIN_S(R) \in S$ implies that such a value always exists.

(ii) $LIN(R) \in S$ for every locally correct R: Local correctness of R implies that R is weakly plausible. By Lemma 5.4, R is strongly plausible. By Lemma 5.3, every linearization of $\overline{R}$ is legal. If $\overline{R}$ is acyclic then such a linearization exists, and $\overline{\mathcal{H}}(R) \neq \emptyset$. Hence $LIN_S(R) \in S$. □

5.2.3 Proving Acyclicity

In Chapter 4 we gave an example of a specification for a simple counter that does not have a CBCAST implementation. This specification is commutative: READ operations are read-only and INC operations commute. However, the acyclicity requirement in Theorem 5.1 is not satisfied as the example in Figure 5.2 shows. The run in this figure is plausible; for example

$$R[Read_1 : 6] = Inc_2(1) \rightarrow Inc_1(5) \rightarrow Read_1 : 6$$

has only one linearization, and this linearization is legal. However the closure of R has a cycle

$$Inc_1(5) \rightarrow Read_1 : 6 \mapsto Inc_3(3) \rightarrow Read_3 : 4 \mapsto Inc_1(5).$$

As we have already seen in Section 4.2.1, this problem has no asynchronous implementation.

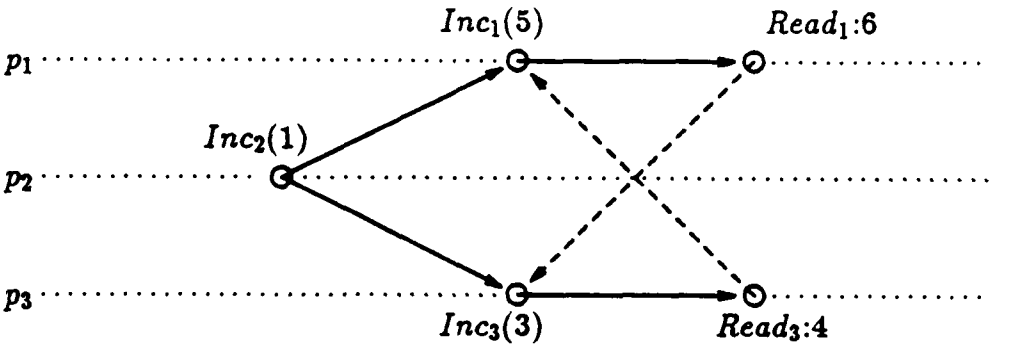


Figure 5.2: An example run

In this section we will present techniques for deciding whether a specification is cyclic or not. We will illustrate our techniques by applying them to our token passing example.

Definition 5.10

Let R be a run with a cycle C in $\overline{R}$:

$$\begin{aligned} C = \; & e_{1,1} \rightarrow e_{1,2} \rightarrow \ldots \rightarrow e_{1,n_1} \\ \mapsto \; & e_{2,1} \rightarrow e_{2,2} \rightarrow \ldots \rightarrow e_{2,n_2} \\ & \ldots \\ \mapsto \; & e_{m,1} \rightarrow e_{m,2} \rightarrow \ldots \rightarrow e_{m,n_m} \\ \mapsto \; & e_{1,1} \end{aligned}$$

We call

$$e_{i,1} \rightarrow e_{i,2} \rightarrow \ldots \rightarrow e_{i,n_i}$$

(for $i = 1 \ldots m$) a segment of the cycle.

Lemma 5.5

Every cycle in the closure $\overline{R}$ of a run R has at least two segments.

Proof: Because R itself is acyclic, every cycle in $\overline{R}$ must contain at least one "$\mapsto$" edge. Consider a cycle with only one segment:

$$C = e_1 \rightarrow e_2 \ \ldots \ \rightarrow e_n \mapsto e_1$$

According to Definition 5.5 $\overline{R}$ contains "$\mapsto$" edges only between events that are concurrent in R. Since $e_1 \rightarrow e_n$ (by transitivity) $\overline{R}$ cannot contain the edge $e_n \mapsto e_1$.

□

Lemma 5.6

If $\overline{R}$ has a cycle then it also has cycle in which

(i) all segments are concurrent, i.e., $a//b$ for any two events a and b in different segments.

(ii) every segment has at most two events.

Proof: (i) Let

$$\begin{aligned} C = \ & e_{1,1} \rightarrow e_{1,2} \rightarrow \ldots \rightarrow e_{1,n_1} \\ \mapsto \ & e_{2,1} \rightarrow e_{2,2} \rightarrow \ldots \rightarrow e_{2,n_2} \\ & \ldots \\ \mapsto \ & e_{m,1} \rightarrow e_{m,2} \rightarrow \ldots \rightarrow e_{m,n_m} \\ \mapsto \ & e_{1,1} \end{aligned}$$

be a cycle in $\overline{R}$. Assume C has two non-concurrent segments. Then there are two events $a = e_{i,j}$ and $b = e_{k,l}$, such that $a \rightarrow b$ in R. We can use this relation to construct a smaller cycle

$$\begin{aligned} C' = \; & a \rightarrow b \rightarrow e_{k,l+1} \rightarrow \ldots \rightarrow e_{k,n_k} \\ & \mapsto e_{k+1,1} \rightarrow e_{k+1,2} \ldots \\ & \ldots \\ & \mapsto e_{i,1} \rightarrow \ldots \rightarrow e_{i,j-1} \rightarrow a \end{aligned}$$

The cycle C' has strictly less segments than C, because C' does not contain

$$\begin{aligned} & a \rightarrow e_{i,j+1} \rightarrow \ldots \rightarrow e_{i,n_i} \\ \mapsto \; & e_{i+1,1} \rightarrow \ldots \rightarrow e_{i+1,i+1} \\ & \ldots \\ \mapsto \; & e_{k,1} \rightarrow \ldots \rightarrow e_{k,l-1} \rightarrow b \end{aligned}$$

which has been replaced by the "short cut" $a \rightarrow b$. We repeat this process until the resulting cycle no longer has non-concurrent segments. Lemma 5.5 ensures that the process need only be repeated a finite number of times.

(ii) Consider a segment

$$e_{i,1} \rightarrow e_{i,2} \rightarrow \ldots \rightarrow e_{i,n_i}$$

that has more than two events. Because of the transitivity of "$\rightarrow$" this segment can be replaced by the shorter segment

$$e_{i,1} \rightarrow e_{i,n_i}.$$

□

This lemma expresses the following intuitive idea: The CBCAST implementation breaks down if two different processors take mutually inconsistent actions without knowing about the others action. The inconsistency of these actions is expressed as a cycle in a run. The fact that the two processors do not know about each other's actions is expressed by the corresponding events being concurrent.

Specifications that are acyclic have the property that certain types of events which are part of ordering constraints can never occur concurrently in a plausible run. We call such events *mutually exclusive.*

Definition 5.11

Two events a and b are mutually exclusive under S if

$$\forall R\colon \quad R \text{ plausible} \quad \Rightarrow \quad \neg\, a // b.$$

We prove a specification to be acyclic by showing that any cycle in the closure of a plausible run would contain mutually exclusive events in different segments of the cycle. This would force all cycles to have non-concurrent segments. However, this contradicts Lemma 5.6, which we just proved.

Let us return to our token passing example. We will now prove that every plausible run of the token passing specification has an acyclic closure.

Theorem 5.2

Consider the token passing example. Two successful PASS events of the form

$$a = P_i(x){:}ok, \text{ and } b = P_j(y){:}ok, \text{ for } i \neq j$$

are mutually exclusive.

This claim is very intuitive. If two such events were not mutually exclusive there could be two processors holding the token at the same time, violating the token passing specification.

Proof: Consider a run R with two concurrent pass events $a = P_i(x){:}ok$ and $b = P_j(y){:}ok$. We show that R cannot be plausible by induction on the number of events in R.

Base case: R contains no events other than a and b. Assume R is plausible. Then $\overline{R[a]} = a$ and $\overline{R[b]} = b$ must have legal linearizations $H_a = (a)$ and $H_b = (b)$ respectively. Because processor 1 is the initial token holder $H_a = (P_i(x){:}ok)$ can only be legal if $i = 1$. For the same reason H_b is only legal if $j = 1$. But $i = j$ contradicts the assumption that a and b are concurrent.

For the induction step consider R with more than two events. Assume R is plausible. Then $\overline{R[a]}$ and $\overline{R[b]}$ have legal linearizations H_a and H_b respectively. Let $R' = R[a] \cap R[b]$. By induction hypothesis $R[a]$, $R[b]$, as well as R' do not have concurrent pass events. Therefore we can define the following events:

$c = P_l(z){:}ok$	The last pass event in R'.
$a' = P_i(x){:}ok$	The first pass event after c in $R[a]$.
$b' = P_j(y){:}ok$	The first pass event after c in $R[b]$.

(where possibly, but not necessarily $a' = a$ and/or $b' = b$.) Note that $a' // b'$ because otherwise either a' or b' would be in R'. Then the histories H_a and H_b have the form

$$H_a = \ldots \; P_l(z){:}ok \; \ldots \; P_i(x){:}ok \; \ldots \; a$$
$$H_b = \ldots \; P_l(z){:}ok \; \ldots \; P_j(y){:}ok \; \ldots \; b$$

with no pass events between c and a' in H_a and between c and b' in H_b. Then H_a can only be legal if $i = z$, otherwise the operation $P_i(x)$ should return an error code eH. For the same reason H_b is only legal if $j = z$. Hence $i = j$, i.e., a' and b' are events at the same processor. But that contradicts $a'//b'$. □

Not only pass operations but any two events that indicate that the caller is the current token holder are mutually exclusive:

Theorem 5.3

Any two events of the following types are mutually exclusive:

$$Q_i{:}T,\ \ R_i{:}eH,\ \ P_i(x){:}ok,\ \text{ or }\ P_i(x){:}eR$$

The proof is very similar to the one for Theorem 5.2; it is carried out in Appendix A.

Now let us consider the ordering constraints occurring in the token passing specification. These constraints are of one of the following three types (see Appendix A):

(I) $Q_i{:}F \quad \mapsto \quad P_j(i){:}ok$

(II) $R_i{:}eR \quad \mapsto \quad P_j(i){:}ok$

(III) $P_i(j){:}eR \quad \mapsto \quad R_j{:}ok$

Theorem 5.4

The token passing specification is acyclic.

Proof: Assume not. Let R be a plausible run that contains a cycle. By Lemma 5.6 we may assume that the cycle only has concurrent segments. Consider the ordering constraint edges ("$\mapsto$") in such a cycle. The cycle cannot contain more than one edge of type (III), otherwise there would be two pass events in different segments of the cycle, which is not possible since segments are concurrent and pass events

are mutually exclusive. For the same reason there cannot be more than one edge of type (I) or (II) in the cycle. By Lemma 5.5 the cycle has at least two "$\mapsto$" edges; hence it must have exactly one edge of type (III) and one of type (I) or (II). Hence the cycle is of the following form:

$$\begin{aligned} C \;=\; & P_j(i){:}ok \;\rightarrow\; P_k(l){:}eR \\ \mapsto\; & R_l{:}ok \;\rightarrow\; e \\ \mapsto\; & P_j(i){:}ok \end{aligned}$$

where either $e = Q_i{:}F$ or $e = R_i{:}eR$.

The first segment of the cycle consists of the two pass events $a = P_j(i){:}ok$ and $b = P_k(l){:}eR$. If R is plausible than $\overline{R[b]}$ has a legal linearization H_b. Because $a \rightarrow b$, a is in $R[b]$ and therefore also in H_b. Hence H_b has the form

$$H_b \;=\; \ldots\; P_j(i){:}ok \;\ldots\; P_k(l){:}eR$$

Notice that the return value eR of the last event (the pass operation failed because processor l did not request the token) indicates that processor k is holding the token at that time. Therefore H_b must contain a pass event $c = P_i(x){:}ok$ between the two events a and b in H_b; otherwise processor i would still be holding the token at the end of H_b. From Theorem 5.3 we know that c cannot be concurrent with a or b; hence

$$a \rightarrow c \rightarrow b.$$

Now consider the event e in the second segment of the cycle. Events c and e cannot be concurrent, because the operations were both invoked at processor i. If $c \rightarrow e$

we have $a \rightarrow c \rightarrow e$; hence $a \rightarrow e$. If $e \rightarrow c$ we have $e \rightarrow c \rightarrow b$; hence $e \rightarrow b$. In both cases the two segments of the cycle would not be concurrent, contradicting Lemma 5.6. □

Let us summarize our techniques for deciding whether a specification is acyclic. Lemma 5.6 allows us to restrict our search for cycles to certain simple types of cycles with the following three properties:

1. The cycle has at least two segments, i.e., it contains at least two "$\mapsto$" edges.

2. Every segment has exactly two events.

3. All segments are mutually concurrent.

Because of properties 1 and 3, such a cycle must have concurrent events that occur in an ordering constraint. Therefore we are successful if we can show that events that are involved in ordering constraints are mutually exclusive, i.e., do not occur concurrently in a plausible run.

5.3 Mixed Implementations

The techniques we outlined in the previous sections are still useful if a specification is cyclic or even if it is not strictly commutative. In the case where these techniques fail to produce a correct asynchronous implementation for a specification S, it is often not necessary to resort to an implementation that is based on atomic broadcasts only. Instead, it is often possible to construct a *mixed implementation* in which most events are propagated with CBCAST and ABCAST is used only for certain "critical" events. For example, consider a service for managing shared data. If clients are required to explicitly acquire locks before modifying the data, then only LOCK and

UNLOCK operations need to be globally ordered. Once a lock is granted the actual updates may be propagated asynchronously [JB86]. The techniques developed in the previous sections of this chapter allow us to identify what events are "critical": events that do not commute and events that occur in cycles.

In this section we will outline how the results from this and the preceding chapter can be generalized to apply to such mixed implementations. We modify our definition of implementation by adding a parameter A defining the set of "critical " operations that must be propagated by an atomic broadcast. That is, an implementation is now a 9-tuple:

$$(n, I, V, M, Q, q_0, \Phi, \Psi, A), \quad \text{where } A \subset I.$$

We also need a new ordering axiom that defines mixed implementation histories:

Definition 5.12

Ordering axiom for mixed implementations:

(i) Causal ordering:

$$inv_E\langle i,j\rangle \rightarrow inv_E\langle l,m\rangle \;\Rightarrow\; \forall k\colon\; rcv_E(\langle i,j\rangle,k) <_k rcv_E(\langle l,m\rangle,k)$$

(iia) $\forall\, inv_E\langle i,j\rangle \in A$: (Global ordering)

$$\forall i',j'\colon\; \forall k,l\colon$$

$$rcv_E(\langle i,j\rangle,k) <_k rcv_E(\langle i',j'\rangle,k) \;\Leftrightarrow\; rcv_E(\langle i,j\rangle,l) <_l rcv_E(\langle i',j'\rangle,l)$$

(iib) $\forall\, inv_E\langle i,j\rangle \in I - A$: (Immediate local delivery)

$$\forall i,j\colon\; \neg\,\exists\, a\colon\; inv_E\langle i,j\rangle <_k a <_k rcv_E(\langle i,j\rangle,i)$$

The axiom requires that all message delivery must be consistent with "$\rightarrow$" (i), that all message delivery must be globally ordered with respect to messages sent by atomic broadcast (iia), and that messages sent by CBCAST are immediately delivered

locally. Notice that if $A = \emptyset$ (all events propagated by CBCAST) this definition is equivalent to the CBCAST ordering axiom (Definition 4.5).

A mixed implementation is constructed the same way as a CBCAST implementation, based on a linearization function. However, the correctness condition can be relaxed, because certain types of runs cannot occur in an execution of a mixed implementation. We formalize this below:

Definition 5.13

A run R is called permissible under A if events with invocations in A are globally ordered with respect to all other events:

$$\forall e \in A \times V: \ \forall e' \in R: \ \neg\, e // e'.$$

Lemma 5.7

Let Y be a mixed implementation and let E be a mixed execution history. Then $R_Y(E)$ is permissible.

Proof: Follows immediately from Definition 5.12 (iia). □

Because of this property of mixed implementation, the correctness condition that we established for CBCAST implementation (Theorem 4.1) needs to be satisfied only for permissible runs:

Theorem 5.5

If LIN is constructive and satisfies

$$\forall \text{ permissible runs } R: \ R \text{ locally correct } \Rightarrow \ LIN(R) \in S.$$

then $Y_{S,LIN}$ is a correct mixed implementation of specification S.

Proof: We show that for every mixed execution history E, the history $H = LIN(R_Y(E))$ is legal and satisfies the correctness and liveness conditions of Definition 3.12.

Let E be a mixed execution history, and let $R = R_Y(E)$. By Lemma 4.5, R is locally correct. By Lemma 5.7, R is also permissible. Therefore, by assumption, the history $H = LIN(R)$ is legal. The rest of the proof is exactly the same as the proof of Theorem 4.1 on page 64. □

This theorem also allows us to generalize the results of Section 5.2.2.

Corollary 5.2

If a commutative specification S satisfies the following condition

$$\forall \text{ permissible } R\text{: } R \text{ plausible } \Rightarrow \overline{R} \text{ acyclic,}$$

then the mixed implementation under LIN_S (Definition 5.6) is correct.

In the previous section we demonstrated how to prove that a specification is acyclic by showing that events that could form a cycle do not occur concurrently in a plausible run (i.e., are mutually exclusive). Events that could form a cycle but are not mutually exclusive cause this technique to fail. However, by Corollary 5.2, a mixed implementation will be correct if such events are propagated by atomic broadcast, ensuring that they do not occur concurrently in a permissible run.

In a similar way it is possible to extend our results to mixed implementations in which events that are not commutative are propagated by atomic broadcast.

5.4 Summary

In this chapter we saw that in general, the question of whether a linearization operator exists for a given specification is undecidable. Hence there are no general methods for finding such operators. Therefore, we considered a restricted class of specifications which we call commutative. We show how to exploit knowledge about the commutativity of events to construct linearization operators for specifications in this class. These methods are useful for developing efficient asynchronous implementations for a broad range of practical problems.

Chapter 6

Failures

In our treatment so far we have assumed a distributed system that is perfectly reliable. However, one of the main uses of broadcast protocols is in the design of fault-tolerant programs. In this chapter we will address the problems that arise if we take processor failures into account.

6.1 Integrating Failures into the Model

In chapters 3 and 4 we showed how to take a formal specification of a centralized service and use broadcast protocols to construct a distributed implementation of this service. Now we want to make the distributed service fault tolerant. What we mean by "fault tolerant" is that even if some processors fail, the behavior of the distributed service should be indistinguishable from a perfectly reliable centralized server. As we will see in this section we achieve this goal simply by replacing the broadcast protocols used in the implementations constructed in Chapters 3 and 4 by reliable versions of the same protocol. In other words, if the broadcast protocol used in an implementation provides atomic message delivery, the implementations

will automatically be fault tolerant.

To be more precise, we must add failures to our execution model. An execution history may now contain *failure events* in addition to invocation and receive events. We modify our definition of execution histories (definitions 3.4 and 3.7) accordingly:

Definition 6.1

An unreliable execution history $E = (E_1, \ldots, E_n)$ is a collection of ordered sets of *invocation*, *receive*, and *failure* events,

$$E \in [(I \cup N^2 \cup \{\text{FAIL}\})^*]^n,$$

satisfying the following conditions:

(i) Reliable message delivery:

$\forall\, inv_E\langle i,j\rangle\colon \ \forall\, k\colon \ \exists$ unique receive event $(i,j) \in E_k$

(ii) Sequential invocation:

$\forall\, i,j\colon \ rcv_E(\langle i,j\rangle, i) <_i inv_E\langle i, j+1\rangle$

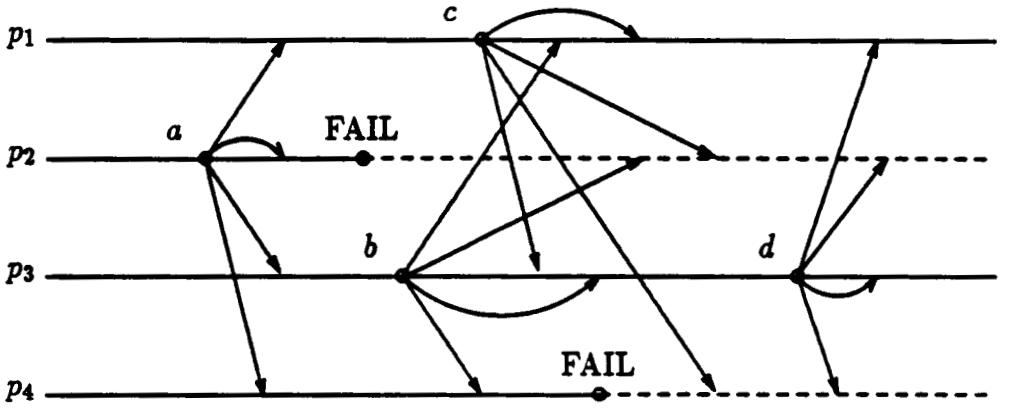
(iii) Monotonicity of time:

$\neg\, \exists\, e_1, \ldots, e_m \in E\colon \ e_1 \xrightarrow{D} e_2 \xrightarrow{D} \ldots \xrightarrow{D} e_m \xrightarrow{D} e_1.$

(iv) No invocation events after a failure:

$\forall\, k\colon$ FAIL $\in E_k \neq \exists$ invocation event $a \in E_k\colon \ a >_k$ FAIL

Conditions (i – iii) are exactly the same as in definitions 3.4 and 3.7. For notational convenience we pretend that a processor that has crashed still receives broadcasts. Hence a failure is simply an event after which a processor stops sending any new messages (condition (iv)). Figure 6.1 illustrates such an execution history. Notice that this model describes an implementation based on *reliable* broadcast protocols,



$E_1 = (2,1)\ c\ (3,1)\ (1,1)\ (3,2)$

$E_2 = a\ \text{FAIL}\ (2,1)\ (3,1)\ (1,1)\ (3,2)$

$E_3 = (2,1)\ b\ (1,1)\ (3,1)\ d\ (3,2)$

$E_4 = (2,1)\ (3,1)\ \text{FAIL}\ (1,1)\ (3,2)$

Figure 6.1: An execution history with failure events

because condition (i) in Definition 6.1 ensures atomic message delivery.

The definition of an implementation as an 8-tuple $(n, I, V, M, Q, q_0, \Phi, \Psi)$ remains unchanged, but we have to specify the effect of failure events. We do so by defining the state of a processor after a failure to be *undefined* ($\perp$), that is we modify the definition of $stat_E[i,j]$ as follows:

Definition 6.2

$$stat_E[i,j] = \begin{cases} q_0 & \text{if } j = 0 \\ \perp & \text{if } E[i,j] = \textbf{FAIL} \\ \perp & \text{if } stat_E[i,j\text{-}1] = \perp \\ \phi_i^s(stat_E[i,j\text{-}1], a) & \text{if } E[i,j] = a \text{ is an invocation event} \\ \psi_i^s(stat_E[i,j\text{-}1], m) & \text{if } E[i,j] = (k,l) \text{ is a receive event, where} \\ & m = \phi_k^m(stat_E[k, inum_E\langle k,l\rangle\text{-}1], inv_E\langle k,l\rangle) \end{cases}$$

The definitions of $msg_E\langle i,j\rangle$, $val_E\langle i,j\rangle$, $event_E\langle i,j\rangle$, $H[E,i]$ remain the same as before (Definition 3.11). In an unreliable system we define an implementation to be correct if all *operational* sites cannot distinguish its behavior from that of a perfectly reliable centralized service:

Definition 6.3

Y is a correct XBCAST-implementation of specification $S = (n, I, V, S)$ iff:

$\forall$ XBCAST execution history E: $\exists\, H \in S$:

Correctness: $\forall\, i$: if FAIL $\notin E_i$ then $H|_i = H[E,i]$

Liveness: $\forall\, i,j,k$:

$$rcv_E(\langle i,j\rangle, k) <_k inv_E\langle k,l\rangle \;\Rightarrow\; event_E\langle i,j\rangle <_H event_E\langle k,l\rangle$$

The fact that in our execution model message delivery is reliable ensures that processors that do not fail are not affected by the failure of other processors. This is expressed in the following lemma:

Lemma 6.1

Let E be an execution history with failure events, and let E' be identical to E except that all failure events are deleted from it. Then

(i) E' is an well formed execution history.

(ii) $\forall i$: if E_i does not contain a failure event then $H[E,i] = H[E',i]$.

Proof: (i) As an unreliable execution history, E satisfies condition (i – iii) in Definition 6.1. Condition (i) ensures that E' is an execution sequence according to Definition 3.4; conditions (ii, iii) ensure that this execution sequence is a well formed execution history (Definition 3.7).

(ii) By construction, all invocations in $H[E,i]$ and $H[E',i]$ are identical. Hence we only have to show that all return values are also the same. Consider the formal event $e = a{:}v = event_E\langle i,j\rangle \in H[E,i]$. Then $a = inv_E\langle i,j\rangle$ and $v = val_E\langle i,j\rangle$. Let $b = E[i, rnum_E(\langle i,j\rangle, i) - 1]$ be the corresponding receive event. Recall that according to Lemma 4.1 $val_E\langle i,j\rangle$ only depends on events that precede b under "$\rightarrow$". Hence we are done if we can show that $E[b] = E'[b]$. Assume that $E[b] \neq E'[b]$. This is only possible if $E[b]$ contains a failure event f. Because a processor does not send any messages after it fails, the only events related to a failure event f are receive events after f at the same processor. Hence $f \in E[b]$ implies that $f \in E_i$ contradicting our assumption that E_i does not contain a failure event. □

Theorem 6.1

Every correct implementation of a specification S is also fault-tolerant.

Proof: Let Y be a correct implementation of S and let E be an unreliable execution history. Let E' be E with failure events deleted. Because Y is correct, there exists a history $H \in S$ that satisfies the correctness and liveness conditions with respect to E'. By Lemma 6.1 $H[E,i] = H[E',i]$ for all E_i with no failure events. Therefore H will also satisfy the correctness and liveness conditions with respect to E as stated in Definition 6.3. □

6.2 Client Failures

A processor failure not only affects a component of a distributed service, but also the client running at that site. The designer of a distributed service may want to explicitly specify a particular action to be taken if a client fails. In the token passing service, for example, it is desirable that the token is not lost if its current holder fails. Hence, we would want to specify the behavior of the token passing service in such a way that the token is automatically transferred to some other client in the case of a failure.

This problem can be solve within our formalism by treating a client failure like any other operation invoked by a client. In other words, the specification is designed as if a client invoked a special operation "CRASH" just before its processor fails. If the distributed system provides a means of detecting failures, such a specification can be implemented in the same way as specifications that do not contain client failures. For example, the ISIS system provides a failure detection and notification

mechanism that makes the failure of a processor look as if the processor sent out a broadcast announcing its death just before the failure [BJ87b,BJSS86].

6.3 Summary

In this chapter we showed that reliable broadcast protocols can be used to construct a fault-tolerant distributed service. This approach is very similar to the method of replicated state machines described by Schneider in [Sch86].

Chapter 7

Conclusion

7.1 Summary and Discussion

We considered a variety of reliable broadcast protocols that differ in the form of message ordering they provide: atomic broadcast (ABCAST), causal broadcast (CBCAST), FIFO broadcast (FBCAST), and unordered broadcast (BCAST). The stronger the ordering property of the protocol the more costly its implementation. There is a fundamental difference between atomic broadcast and the other forms of broadcasts. An atomic broadcast protocols requires at least two phases of message exchange, whereas CBCAST, FBCAST, and BCAST can be implemented as one-phase protocols. Furthermore, in an unreliable system in which processors may experience failures ABCAST can only be implemented if failures are detectable or if an upper bound on message delays is known. CBCAST, FBCAST, and BCAST, on the other hand, can be implemented reliably in a completely asynchronous system.

Our results from Chapter 4 show that this fundamental difference is also reflected in the classes of problems that can be solved with a particular broadcast protocol. We showed that the class of all formal specification separates into two distinct

subclasses $\mathcal{S}_{async}$ and $\mathcal{S} - \mathcal{S}_{async}$, which correspond to specifications that have an implementation based on CBCAST, FBCAST, or BCAST, and specifications that require the global ordering that ABCAST provides.

For specifications in $\mathcal{S}_{async}$, an implementation can be expressed in a canonical form based on a linearization function for that specification. Although the existence of such a function in general is undecidable, it is possible to analyze commutativity and dependencies between events to find linearization functions for a subclass of $\mathcal{S}_{async}$. The methodology introduced in Chapter 5 allows identification of conflicting events and establishes conditions that allow the construction of an asynchronous implementation. Specifications for which this method is successful could be characterized as "self-synchronizing", that is the specification itself prevents certain conflicting events from occurring concurrently. Even if our techniques fail to yield a completely asynchronous implementation they are still useful for constructing a mixed implementation, as they identify a subset of events that need to be propagated by atomic broadcast.

7.2 Future work

It should be possible to extend the results from Chapter 5. Notice that the conditions presented in that chapter are sufficient but not necessary for the existence of an asynchronous implementation. This naturally raises the question of whether the methodology can be generalized to cover a larger set of specifications. For example, there are non-commutative specifications that have asynchronous implementations. Examples are specifications in which only operations invoked by one particular processor are sensitive to the order of the events. For example, one can modify the token passing specification to require token requests to be serviced in FIFO order.

Such a specification is not commutative, because token requests no longer commute. However, because only the current token holder decides which processor receives the token next, it is not necessary that token requests are globally ordered.

Another interesting problem is to generalize our formalism to allow implementations that exhibit "temporary inconsistencies". In Chapter 4 we showed that the problem of implementing a shared counter does not have an asynchronous solution. However, for certain types of implementations it might be acceptable if a read operations returns the sum of only a subset of previous increments, as long every increment is *eventually* reflected in all future reads. The formalism presented in Chapter 3 allows us to relax the shared counter specification to allow reads to return partial sums. However, because we need specifications to be prefix-closed, we cannot express the requirement that increments are not ignored forever. One way of solving this problem might be to define specifications as sets of partially ordered sets of events (i.e., runs) rather then sets of histories. One could then specify a shared counter in such a way that a read operation is allowed to ignore an increment only if it is concurrent to the read. The drawback of this approach is that specifications no longer have the intuitive meaning of ensuring that the distributed program behaves behaves like a centralized server.

Appendix A

An Example: Token Passing

A.1 Formal Specification

We want to implement a distributed token passing algorithm. The client interface consists of the following three operations:

- QUERY(): BOOLEAN

  — *returns* TRUE *if the caller is the current token holder.*

- PASS(X: CLIENTID): RETURNCODE

  — *passes the token from the current token holder to client* x.

  This operation returns one of three values: OK, ERRORHOLDER (the caller is not the current token holder), or ERRORREQUEST (client x did not request the token).

- REQUEST(): RETURNCODE

  — *request the token.*

This operation returns one of three values: OK, ERRORHOLDER (the caller is already holding the token), or ERRORREQUEST (the caller has already requested the token).

We use the following abbreviated notation for operations and return values:

Q	QUERY
P	PASS
R	REQUEST
T	TRUE
F	FALSE
eH	ERRORHOLDER
eR	ERRORREQUEST

Given a formal history H, we identify the current token holder, $CurHold(H)$, to be the client that token was last passed to, where client 1 is the initial holder of the token:

$$CurHold(H) = \begin{cases} 1 & \text{if } H \text{ does not contain any successful PASS operations.} \\ x & \text{if the last successful PASS event in } H \text{ has} \\ & \text{the form } P_i(x){:}ok, \text{ for some } i. \end{cases}$$

We can further define a predicate that tell us whether there is a pending token request by a particular client:

$$PendReq(H, x) = \begin{cases} \text{TRUE} & \text{if } R_x{:}ok \in H \text{ and if } H \text{ does not contain} \\ & \text{an event } P_i(x){:}ok \text{ after this request.} \\ \text{FALSE} & \text{otherwise.} \end{cases}$$

A formal specification S for our token passing example is given by the following recursive definition:

1. $\emptyset \in S$
2. $\forall H \in S$: let $x = CurHold(H)$:
   (i) $\forall y \neq x$: $H + Q_y{:}F \in S$.
   and $H + Q_x{:}T \in S$.
   (ii) $\forall j$: if $PendReq(H, j)$ then $H + P_x(j){:}ok \in S$.
   if $\neg PendReq(H, j)$ then $H + P_x(j){:}eR \in S$.
   (iii) $\forall y \neq x$: $\forall j$: $H + P_y(j){:}eH \in S$.
   (iv) $\forall y \neq x$: if $\neg PendReq(H, y)$ then $H + R_y{:}ok \in S$.
   if $PendReq(H, y)$ then $H + R_y{:}eR \in S$.
   (v) $H + R_x{:}eH \in S$.
3. S is the smallest set satisfying the above.

The specification S says that the QUERY should always return TRUE to the current token holder, and that only the current holder is allowed to pass the token to any other client. This specification describes an idealized token passing system, in the sense that passing a token is supposed to be an instantaneous event: A PASS operation takes effect immediately, because any operation following a PASS is required to reflect the new token holder. Fortunately our definition of implementation correctness only requires that the behavior of the system is indistinguishable from this idealized behavior. To illustrate this point, consider a system with only two processors. A PASS operation may be implemented by simply sending a message from the previous to the new token holder. An external observer may see the following history:

$$H = (\ P_1(2),\ \ Q_2{:}F,\ \ Q_2{:}T\ )$$

Obviously the pass message was delayed a little so that only the second QUERY operation returned client 2 as the current token holder. Although $H \notin S$ we still consider the implementation correct, because, to the clients, H is *indistinguishable* from the legal history

$$H = (\ Q_2{:}F,\ \ P_1(2),\ \ Q_2{:}T\ ).$$

A.2 Commutativity and Ordering Constraints

Next we apply our theory from Chapter 5 to show that the token passing example indeed has a CBCAST implementation. We start by computing a table of dependencies (Table A.1) between events. There are two pairs of events which are completely interchangeable and need not be considered separately:

$$R_i{:}eH \equiv Q_i{:}T, \text{ and}$$
$$P_i(x){:}eH \equiv Q_i{:}F.$$

For this reason these events share the same row and column in Table A.1.

The table allows us to verify that the token passing specification is commutative. There are two types of update events:

$$R_i{:}ok, \text{ and } P_i(j){:}ok.$$

According to Definition 5.3 we have to show that

$$\forall H{:}\ \ \forall a, b \text{ update events at different processors:}$$
$$Ha \in S \ \wedge\ Hb \in S \ \Rightarrow\ Hab \in S \ \wedge\ Hba \in S \ \wedge\ Hab \equiv Hba.$$

Table A.1 shows that for any two such update events a and b, either the two events commute (o entry in the table), or there is no H such that Ha and Hb are both legal ($\times$ in the table).

Table A.1: Dependencies between events in the token passing specification

	$Q_j : T$ R_j:eH	$Q_j : F$ $P_j(y)$:eH	$R_j : eR$	$P_j(y) : eR$	$R_j : ok$	$P_j(y) : ok$
$Q_i : T$ R_i:eH	×	o	o	×	o	×
$Q_i : F$ $P_i(x)$:eH	o	o	o	o	o	↦ $y = i$ o $y \neq i$
$R_i : eR$	o	o	o	o	o	↦ $y = i$ o $y \neq i$
$P_i(x) : eR$	×	o	× $x = j$ o $x \neq j$	×	↦ $x = j$ o $x \neq j$	×
$R_i : ok$	o	o	o	o	o	× $y = i$ o $y \neq i$
$P_i(x) : ok$	×	o	o	×	× $x = j$ o $x \neq j$	×

o **The events commute, i.e., $Ha \equiv Hb$, for all H.**

× **The events are incompatible, i.e., there does not exist any history H such that Ha and Hb would both be legal.**

↦ **There is an ordering constraint between the two events (Definition 5.4).**

A.3 Mutually Exclusive Events

Next, we need to show that every plausible run has an acyclic closure. We exploit the fact that certain types of events are mutually exclusive. These are all events that indicate that its caller is currently holding the token.

Definition A.1

$$B_i \;=\; \{Q_i : T, R_i{:}eH\} \cup \{P_x(i){:}ok \mid \text{ for all } x\} \cup \{P_i(x){:}eR \mid \text{ for all } x\}$$

Lemma A.1

The set B_i contains all events that indicate that processor i is holding the token when the event occurs, i.e.,

$$a \in B_i \;\wedge\; Ha \in S \;\Rightarrow\; \mathit{CurHold}(H) = i.$$

Proof: Follows immediately from the token passing specification and the definition of B_i. □

Lemma A.2

Let $B = \bigcup_{i=1\ldots n} B_i$.

All events in B are mutually exclusive.

The proof of a restricted form of this lemma was already presented in Section 5.2.3.

Proof: Consider a run plausible run R with two events $a \in B_i$ and $b \in B_j$. We have to show that a and b cannot be concurrent. We do this by induction on the number of events in R.

Base case: R contains no events other than a and b. Because R is plausible $\overline{R[a]} = a$

and $\overline{R[b]} = b$ have legal linearizations $H_a = (a)$ and $H_b = (b)$ respectively. By Lemma A.1, $H_a = (a) \in S$ and $a \in B_i$ imply that $i = CurHold(\emptyset) = 1$. For the same reason $j = CurHold(\emptyset) = 1$. Therefore $a, b \in B_1$. Hence a and b cannot be concurrent, because all events in B_1 correspond to operations invoked by the same processor (processor 1).

For the induction step consider R with more than two events. Because R is plausible $\overline{R[a]}$ and $\overline{R[b]}$ have legal linearizations H_a and H_b respectively. Let $R' = R[a] \cap R[b]$. By induction hypothesis $R[a], R[b]$, as well as R' do not contain any concurrent events from the set B. Therefore we can define the following events:

$c = P_l(z){:}ok$	The last event in $R' \cap B$.
$a' = P_i(x){:}ok$	The first event after c in $R[a] \cap B$.
$b' = P_j(y){:}ok$	The first event after c in $R[b] \cap B$.

(where possibly, but not necessarily, $a' = a$ or $b' = b$). Note that $a' // b'$ because otherwise either a' or b' would be in R'. Then the histories H_a and H_b have the form

$$H_a = \ldots \; P_l(z){:}ok \; \ldots \; P_i(x){:}ok \; \ldots \; a$$
$$H_b = \ldots \; P_l(z){:}ok \; \ldots \; P_j(y){:}ok \; \ldots \; b$$

with no pass events between c and a' in H_a and between c and b' in H_b. Then H_a can only be legal if $i = z$, otherwise the operation $P_i(x)$ should return an error code eH. For the same reason H_b is only legal if $j = z$. Hence $i = j$, i.e., a' and b' are events at the same processor. But that contradicts $a' // b'$. □

A.4 Acyclicity

We already proved the token passing specification to be acyclic in Section 5.2.3 of Chapter 5. For the sake of completeness we repeat this proof here.

Theorem A.1

The token passing specification is acyclic.

Proof: Assume not. Let R be a plausible run that contains a cycle. By Lemma 5.6 we may assume that the cycle only has concurrent segments. Consider the ordering constraint edges ("$\mapsto$") in such a cycle. The cycle cannot contain more than one edge of type (III), otherwise there would be two pass events in different segments of the cycle, which is not possible since segments are concurrent and pass events are mutually exclusive. For the same reason there cannot be more than one edge of type (I) or (II) in the cycle. By Lemma 5.5 the cycle has at least two "$\mapsto$" edges; hence it must have exactly one edge of type (III) and one of type (I) or (II). Hence the cycle is of the following form:

$$\begin{aligned} C \;=\; & P_j(i){:}ok \;\rightarrow\; P_k(l){:}eR \\ \mapsto\; & R_l{:}ok \;\rightarrow\; e \\ \mapsto\; & P_j(i){:}ok \end{aligned}$$

where either $e = Q_i{:}F$ or $e = R_i{:}eR$.

The first segment of the cycle consists of the two pass events $a = P_j(i){:}ok$ and $b = P_k(l){:}eR$. If R is plausible than $\overline{R[b]}$ has legal linearization H_b. Because $a \rightarrow b$, a is in $R[b]$ and therefore also in H_b. Hence H_b has the form

$$H_b \;=\; \ldots\; P_j(i){:}ok \;\ldots\; P_k(l){:}eR$$

Notice that the return value eR of the last event (the pass operation failed because processor l did not request the token) indicates that processor k is holding the token at that time. Therefore H_b must contain a pass event $c = P_i(x)$:ok between the two events a and b in H_b; otherwise processor i would still be holding the token at the end of H_b. From Theorem 5.3 we know that c cannot be concurrent with a or b; hence

$$a \rightarrow c \rightarrow b.$$

Now consider the event e in the second segment of the cycle. Events c and e cannot be concurrent, because the operations were both invoked at processor i. If $c \rightarrow e$ we have $a \rightarrow c \rightarrow e$; hence $a \rightarrow e$. If $e \rightarrow c$ we have $e \rightarrow c \rightarrow b$; hence $e \rightarrow b$. In both cases the two segments of the cycle would not be concurrent, contradicting Lemma 5.6. □

Appendix B

Invocation-Completion Model

In our formal specifications we consider the execution of an operation to be one single event. This does not allow us to model operations that explicitly wait for another client to take some action (i.e., invoke another operation). Such wait-semantics operations can be modeled if we treat the invocation and the completion of an operation as two separate events.

We argue that it is not necessary to do this: A specification with separate invocation and completion events can be transformed into an equivalent one-operation-one-event specification in which wait-semantics operations are implemented by a busy wait. This works as follows:

Say specification S has an operation A with separate events for the invocation and completion of A (INVOKEA(...) and COMPLETEA:v). We transform S into S' by splitting A into two parts:

STARTA() and QUERYA().

We then make the following two modifications to S:

1. Replace all invocation events INVOKEA(...) by an event STARTA(...):NIL. Replace all completion events COMPLETEA:v by an event QUERYA():v.

2. Add additional histories to S that are obtained by inserting extra QUERYA events to existing histories: Insert an event QUERYA():PENDING anywhere between STARTA(...):NIL and QUERYA():v; insert an event QUERYA():DONE anywhere after QUERYA():v but before the next STARTA(...):NIL.

Then the effect of a client invoking A in S is the same as the client invoking STARTA in S' and then doing a busy wait

while QUERYA() = PENDING do nothing.

With this transformation, an implementation that satisfies the modified specification S' will be equivalent to one that satisfies the original specification S.

Bibliography

[BD87] Ö. Babaoğlu and R. Drummond. (Almost) no cost clock synchronization. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, pages 42–47, Pittsburgh, Pennsylvania, July 1987.

[BG81] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13(2):185–221, June 1981.

[BJ87a] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Eleventh Symposium on Operating System Principles*, pages 123–138. ACM SIGOPS, November 1987.

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

[BJS88] K. Birman, T. Joseph, and F. Schmuck. *ISIS — A Distributed Programming Environment, User's Guide and Reference Manual.* The ISIS Project, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, March 1988.

[BJSS86] K. Birman, T. Joseph, F. Schmuck, and P. Stephenson. Programming with shared bulletin boards in asynchronous distributed systems. Technical Report TR 86-772, Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, August 1986. Revised December 1986.

[CASD84] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. Technical Report RJ 4540 (48668), IBM, October 1984.

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

[FLP85] M. Fisher, N. Lynch, and M. Paterson. Imposibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Had84] V. Hadzilacos. *Issues of fault tolerance in concurrent computations.* Ph.D. dissertation, Harvard University, June 1984. Available as Technical Report 11-84.

[JB86] T. Joseph and K. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, February 1986.

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

[LL86] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, volume ACM 0-89791-198-9/86/0800-0029, pages 29–39. ACM SIGACT-SIGOPS, August 1986.

[LMS85] L. Lamport and P. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Pap79] C. Papadimitriou. Serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[Pet87] L. Peterson. Preserving context information in an IPC abstraction. In *Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 22–31, March 1987.

[SA85] F. B. Schneider and B. Alpern. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Sch85] F. Schmuck. Software clocks and the order of events in a distributed system. Unpublished manuscript, November 1985.

[Sch86] F. B. Schneider. The state machine approach: A tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, December 1986.

[SD83] R. Strong and D. Dolev. Byzantine agreement. In *Proceedings of COMPCON Spring 83*, 1983. San Francisco 1983.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, 1983.

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

[Tan81] A. Tanenbaum. *Computer Networks.* Prentice-Hall Software Serires. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1981.