The Inherent Cost of Achieving Causal Consistency

Carol M. Critchlow
Ph.D Thesis

92-1284
May 1992

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

THE INHERENT COST OF ACHIEVING CAUSAL CONSISTENCY

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

Carol M. Critchlow

August 1991

The Inherent Cost of Achieving Causal Consistency

Carol M. Critchlow, Ph.D.

Cornell University 1991

We consider the problem of distinguishing causally-consistent global states in asynchronous distributed systems. Such states are fundamental to asynchronous systems, because they correspond to possible simultaneous global states; their detection arises in a variety of distributed applications, including global checkpointing, deadlock detection, termination detection, and broadcasting. A *consistent-cut protocol* is a protocol which in every run will designate for each processor a state, in such a way that these states together form a consistent cut. We analyze the cost of achieving causal consistency in terms of the extent to which a consistent-cut protocol delays events of the underlying system, and the message-complexity required by any such protocol. We refer to the delaying action of a protocol as *inhibition.* We consider a spectrum of protocol capabilties based on the type of inhibition that occurs: we distinguish *local* versus *global* inhibition and prove fundamental relationships between these concepts and the ability to determine causally-consistent states. A protocol using local inhibition may cause the delay of some of a processor's events until that processor has performed some number of local actions; a protocol using global inhibition may

force the delay of some of a processor's events until that processor has received some communication from other processors. Based on a variety of system and protocol characteristics, including the ability to locally or globally inhibit particular types of events, we give several impossibility results and examine some existing protocols. We are then able to present a thirty-six-case summary of protocols and impossibility results for the determination of causally-consistent states as a function of those characteristics. In particular, we demonstrate that local inhibition is necessary and sufficient to solve this problem for general FIFO systems, while global send inhibition is necessary and sufficient for general non-FIFO systems. Regarding message complexity, we demonstrate that a globally inhibitory consistent-cut protocol requires $O(N)$ messages where N is the number of processors in the system. This is true whether the protocol is designed to work for FIFO or non-FIFO systems; the exact lower bounds, however, differ in the two cases. We also prove that a consistent-cut protocol which uses local inhibition only requires $O(N^2)$ messages, or, more precisely, $O(|\mathcal{C}|)$, where $\mathcal{C}$ is the channel set of the system. This latter result illustrates a trade-off between the message complexity of a consistent-cut protocol, and the degree of inhibition which it requires.

Biographical Sketch

Carol Critchlow was born on April 19th, 1964, in Vancouver, British Columbia. Her sojourn on the west coast was of short duration, however, and the majority of her childhood was passed in the province of Quebec. Having little sense of direction at the time of her graduation from high school, Carol was tempted south of the border by the lure of a liberal arts education. She entered Amherst College in September, 1981, and spent four years dabbling in subjects ranging from basic electronics to ancient Greek. She received her B.A. in May, 1985. Her career interests still being in soft focus ("maybe pure math, or computer science, or how about operations research?"), she was drawn by the variety of opportunities offered by the Center for Applied Mathematics at Cornell University. Consequently, she started graduate school at Cornell in the fall of 1986, and eventually settled into the study of computer science. Despite the development of an all-consuming addiction to ice hockey during her third year in Ithaca, Carol earned her M.S. in January, 1990, and her Ph.D. in August, 1991. September of 1991 will see her taking up the position of assistant professor in mathematics and computer science at Hobart and William Smith Colleges in Geneva, New York; she hopes it will see her taking up the position of center on the William Smith hockey team as well.

Acknowledgements

First, I would like to propose a toast to Prakash Panangaden, whose encouragement and support—academic, moral, and vinous—have been simply invaluable to me throughout my time at Cornell. If not for him, I *really might* be a substitute teacher of remedial algebra for high school students in Winnipeg. So here's to you, Prakash, and *merci bien*, eh?

I also owe much to Kim Taylor, whose work first introduced me to the world of consistent cuts, and with whom I enjoyed a brief but rewarding collaboration. I am indebted to Bard Bloom for his time and attention; to Al Schatz for doing time on my committee; and to Dolores Pendell for services too numerous to record.

Many, many, *many* blessings upon Anne Rogers and Laurie Hendren, who dragged me—kicking and screaming—to my first hockey practice, and thereby created a monster. It is hard to imagine now what the past few years would have been like without C.S. hockey, and all the activities to which it somehow led: euchre and lacrosse, bridge and Chinese food, highway hockey and casual hockey, pignapping and poetry wars. My heartfelt thanks to all the roommates, teammates, friends, and swine who have shared with me these delights, and have thus brightened my days at Cornell.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Distributed Systems

A distributed system is a network of autonomous processors which communicate with each other from time to time, to exchange information or perhaps perform a system-wide task. Examples of distributed systems include the network of many computers at various different sites which are connected by the INTERNET. The individual machines are all autonomous—there is no central machine directing all the others—but users at the various sites may wish to acquire information that is stored elsewhere. The INTERNET allows users at different machines to communicate with each other, and to transfer information between machines. On a smaller scale, any local area network of workstations is a distributed system. A typical process-control system or parallel machine are also examples.

Attempting to design protocols for such systems gives rise to difficulties which do not occur in a strictly sequential setting: for example, problems of synchronization, contention for resources, or liveness. Various formalisms have been developed to aid in reasoning about these concerns. These include Calculus of

Communicating Systems (CCS) [Mil80], Communicating Sequential Processes (CSP) [Hoa78,Hoa85], Petri Nets [Pet66,Bra80], I/O Automata [LT89], Event Structures [Win86], and several other algebraic or logical formalisms. Different models reflect different possibilities in the properties of a distributed system. For instance, the configuration of the processor network may be static or dynamic. Processors may communicate by message-passing, or by means of shared memory, or a combination of the two. The most fundamental difference is in the degree of synchrony present: processors may operate in lock-step or at arbitrarily different rates; in a message-passing system, communication may be done by means of broadcasts or on a processor-to-processor basis, and there may or may not be upper bounds on the time messages take to be delivered. The status of a single one of these or other variables can effect the solvability of basic problems in distributed systems. The classic example is that of *distributed consensus*: the processors in a system attempt to agree upon the value of a particular variable. Solutions to the problem were sought that would be correct in the presence of some given number of processor failures. It was shown [DDS87] that by varying the assumptions about system synchronization properties, the problem ranges from solvable in the presence of arbitrarily many failures to unsolvable in the presence of even one.

1.2 Asynchronous Systems

The work in this thesis applies to systems in which the processor network is static, processors communicate only via passing messages along point-to-point communication channels, and both the processors and the communication are *asynchronous*. In some distributed systems there may be means for processors to attain some degree of synchrony in their actions. There may, for instance, be a global clock to which all processors have access, and which therefore allows them to perform actions at the same time or in some specific order relative to each other. Alternatively, processors may have separate clocks, but with a known upper bound on the difference in the rates at which the clocks proceed. There may also be a known ceiling on the length of time it takes messages between processors to reach their destination. All of these things aid the processors in the system when it is necessary for them to take some kind of co-operative action. In an asynchronous system, such features are not present. That is: there is no global clock; processors proceed at arbitrarily different rates; and there is unbounded delay in the transmission of messages between processors.

In such a system, it is difficult to obtain a measure of when events at different processors occur in relation to each other. One cannot determine exactly which events occur at any given real time, and consequently one cannot determine the global configuration of the system at any point in its execution. This makes it difficult to discuss what properties do or do not hold of a system at a given real time. Essentially, an alternative to real time is needed in which to frame questions about asynchronous systems. Such a framework was provided by Lamport [Lam78] in the form of *potential causality*, which we discuss in the following section.

1.3 Causality and Consistency

In his work on logical clocks, Lamport introduces the notion of *potential causality.* Two events in an execution of a system are related under potential causality if the occurrence of one could possibly influence the occurrence of the other. In our systems, this could happen in one of three ways.

- The two events occur at the same processor; the occurrence of the earlier one could effect the occurrence of the later one.
- One event is the send of a message, and the second is the reception of the same message.
- The two events fall in the transitive closure of the two relationships above.

If two events on separate processors are not related in one of these ways, then no effect of one event could reach the other processor in time to influence the occurrence of the second event. The relationship is called "happens-before", and it is a subset of temporal precedence: one event happens-before a second only when we can be sure that the first occurred at an earlier time.

In our model of a distributed system, the state of a processor at any given time is represented by the sequence of events that have occurred at that processor up to that time. The state of a system is then represented by a vector of event sequences, one per processor, corresponding to the processors' states at that time. A real-time global state has the property that if a particular event is contained in it, then all events which happened at an earlier time are also included. Having replaced real time with potential causality in the study of asynchronous systems, one considers sets of states with the corresponding property: if a particular event is contained in the set, then all events which happen-before it are also included. Such a set of states is a *consistent global state* or *consistent cut.* It is consistent in the sense that it represents a configuration in which the system could find itself; it is a possible real-time global state. The chief characteristic of such a set of states is that it contains no messages which are received but not sent. In terms of possible real-time states, it would be impossible for a message to be received before it is sent. In terms of potential causality, this would violate the closure under the happens-before relation: a send event happens-before its corresponding receive, so a receive event cannot be included unless the corresponding send event is also.

1.4 Protocols and Inhibition

A *protocol* for a distributed system specifies for each processor a set of events, and an algorithm to determine the order in which that processor should perform those events. The protocol events will occur interspersed among the events that occur at a processor during the course of a normal execution of the system. The protocol should run "on top of" a given system in the sense that it should not prevent the system from executing some sequence of system steps that it would have done in the absence of the protocol. That is, the subsequence of system events in a sequence of system and protocol events should be a valid event sequence of the original system. A protocol may, however, have the power to temporarily obstruct the normal functioning of a processor by forcing it to wait on the occurrence of some particular protocol event before continuing with its task. This is termed *inhibition.* We distinguish two kinds of inhibition: in the first, a protocol only delays events until some number of local actions—sends and internal events—have been performed; in the second, it delays events while a processor waits for communication from another processor. We distinguish these as *local* inhibition versus *global* inhibition.

A *consistent-cut protocol* or *CCP* is a protocol that, in every combined system-protocol run, will distinguish a set of local states that, taken together, form a consistent cut. This in effect means guaranteeing that if the reception of a particular message is included in the designated set of states, then so is the corresponding send. Consistent-cut protocols have numerous distributed applications such as system checkpointing [Rus77,CL85,KT87], deadlock detection [BT87], distributed termination [Fra80], and broadcasting [BJ87].

Chandy and Lamport give in [CL85] an example of a consistent-cut protocol for FIFO systems; that is, systems in which messages sent directly between processors are received in the same order in which they are sent. Each processor in the distributed system will either

1. spontaneously record its state, and then send a marker along each channel incident to it in the network before sending further messages along that channel; or

2. upon receiving a marker, if it has not yet recorded its state, it will do so and send out markers as above.

It is proved in [CL85] that the set of states recorded in the course of the execution of this protocol do indeed form a consistent cut (we present this argument in Chapter 3). Our reason for introducing the protocol here is that it illustrates many of the points which we will discuss in our analysis of consistent-cut protocols. Note first that this protocol does indeed involve inhibition: there is a period at each processor during which that processor cannot send out system messages. There is never any restriction on the reception of messages. One of the issues that we will consider is whether sends, receives, or both must be delayed in order to guarantee the existence of a consistent-cut protocol. A second point to note is that the inhibition is local, as defined above: a processor is prohibited from sending system messages only until it has sent out a special marker; it does not need to wait upon communication from another processor. Again, the question of whether local inhibition suffices in all circumstances is one that we will address in our study of CCPs. Thirdly, this protocol does not work in systems in which the communication is not first-in-first-out (FIFO). In this case, the messages sent by a processor along a channel after the marker (and hence after the processor has recorded its state) may arrive at the destination processor before the marker does (and hence possibly before that processor has recorded its state). We therefore include amongst the issues to be taken into consideration the question of whether or not a system is FIFO. Note that some of the markers themselves are inconsistent: the first marker to reach a given processor is sent after the sending processor has recorded its state; unless the receiving processor has spontaneously initiated the protocol, the marker's reception will be contained in its recorded state. Thus, it is actually only the set of non-protocol events which is consistent. Whether or not protocol messages are allowed to be inconsistent with respect to the cut designated by the protocol will effect the existence of CCPs as well. Finally, the protocol above requires at least one marker between every pair of connected processors. We will examine the lower bounds on the number of messages required by CCPs, and the connection between inhibition and message complexity.

1.5 Summary of Results

We analyze first the inhibitory cost of consistent-cut protocols. The variables discussed above produce thirty-six possible combinations of system properties and protocol characteristics. We prove the following results.

- Using up to global receive inhibition and local send inhibition, there is no CCP for non-FIFO systems. Moreover, this is true even if protocol messages may be inconsistent.

- There is no non-inhibitory CCP for FIFO systems, even if protocol messages may be inconsistent.

- There is no CCP for FIFO systems that uses only local send inhibition, if protocol messages must be consistent.

We also present three well-known protocols, at least one of which pertains to each of the thirty-six cases in which an impossibility is not implied by the above three results.

We also prove the following results concerning the number of messages required by a protocol in order to achieve causal consistency:

- Any consistent-cut protocol for non-FIFO systems requires $O(N)$ messages, where N is the number of processors in the system. More specifically, $2(N-1)$ messages are required.

- Any globally inhibitory consistent-cut protocol for FIFO systems requires $O(N)$ messages. It may require as many as $2N-3$ in the worst case.

- Any locally inhibitory consistent-cut protocol for FIFO systems requires $O(N^2)$ messages. More specifically, it requires $O(|\mathcal{C}|)$ where $\mathcal{C}$ is the channel set of the system.

These last two results illustrate a trade-off between the cost of a protocol in terms of message complexity, and the cost in terms of the degree of inhibition required.

1.6 Outline

The remainder of this thesis is organized as follows. Chapter 2 contains a description of our system model, including formal definitions of the happens-before relation and of causally-consistent cuts. In Chapter 3, we formally define protocols and consistent-cut protocols, and present three well-known examples of CCPs. Chapter 4 contains the formal definition of inhibition, and a discussion of the varying kinds and degrees of inhibition which together form the inhibition spectrum. The four subsequent chapters contain our impossibility and lower bound theorems. We start in Chapter 5 with a set of preliminary lemmas which are used in proving both sets of results. Chapters 6 and 7 contain our impossibility results, the non-FIFO case being discussed in the former, and the FIFO case in the latter. The lower bound results appear in Chapter 8. Finally, we present our conclusions in Chapter 9.

Chapter 2

System Model and Definitions

We describe here a model of asynchronous distributed systems. A *distributed system* consists firstly of a set of autonomous processors, and secondly of a communication medium which allows these processors to exchange information and to co-operate in the achievement of various system-wide tasks. We are interested here in distributed systems in which the processors communicate only by sending and receiving messages along bi-directional channels; there are no shared data structures. By *asynchronous*, we mean that (1) there is no global system clock, (2) there is finite but unbounded delay in the transmission of messages between processors, and (3) processors may proceed at different rates. We assume that the network of processors is connected, though not necessarily completely connected. We also assume that neither the processors themselves nor the communication network can fail. In particular, we assume that any message sent by one processor is eventually received by the processor for which it was intended; any message received by one processor was indeed sent by some other processor; and no message is altered in transmission. Messages in the system are unique, so there can be no confusion as to whether a particular receive event corresponds to a given send event.

Our model is similar to other models of asynchronous systems [CM85,PT88]: the system behaviour is represented by a set of possible runs; each run is composed of sequences of events corresponding to the local actions of each processor, together with a partial order that models potential causality. This notion of a run differs from that of a linearly ordered run, for example the timed runs of [HM90]; a detailed discussion of asynchronous runs is contained in [PT89]. Unlike [CM85,PT88], we explicitly model the *enabling* of events by preceding event sequences, as is typically done in studies of transition systems or event structures [Win86,PT89,LT89,Bra80]. We also explicitly separate protocol and system events (Chapter 3). These two differences will allow us to define *inhibition* in Chapter 4 by comparing the enabling of events by a system alone with the enabling of events by system and protocol together.

2.1 Basic Definitions

Our formal model for distributed systems will consist of five components:

- a processor identifier set and a channel set which describe the physical configuration of the system, which is assumed to be static.
- a description of the individual events which may occur at each processor;
- a description (called an *enabling relation*) of the manner in which events may occur at each processor; and
- a set of runs which describe the possible executions of a system.

We discuss each of these below.

2.1.1 The Physical Structure

A distributed system consists of a set of N processors and a connecting network of channels by means of which the processors communicate.

Definition 1. If $\mathcal{I} = \{1, 2, \ldots, N\}$ is a set of natural numbers used to identify the processors in a system, then $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{I}$ is a *channel set* if $(i, i) \notin \mathcal{C}$ for any $i \in \mathcal{I}$.

The channel set of a system describes the physical links between processors. If i and j are members of $\mathcal{I}$ and (i, j) is an element of $\mathcal{C}$, then processor i can send messages to processor j in the processor network described by $\mathcal{C}$. The restriction $(i, i) \notin \mathcal{C}$ states that a processor does not send messages to itself via the communication network.

We are interested in modeling systems in which the channels are bi-directional: that is, whenever processor i can send messages to processor j, it is also possible for j to send messages to i.

Definition 2. A channel set $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{I}$ is *bi-directional* if whenever $(i, j) \in \mathcal{C}$ for $i, j \in \mathcal{I}$, then $(j, i) \in \mathcal{C}$.

A second feature of the channel sets of the systems which we wish to consider is *connectivity.* We will assume that, for any two processors i and j in the system, there is a communication path from i to j.

Definition 3. The channel set $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{I}$ is *connected* if for every $i, j \in \mathcal{I}$, there exist $i_1, i_2, \ldots, i_n \in \mathcal{I}$ such that $(i, i_1), (i_1, i_2), \ldots, (i_{n-1}, i_n), (i_n, j) \in \mathcal{C}$.

We are often interested in referring to pairs of processors in a network which are capable of exchanging messages directly without necessity of intermediate processors. Since we will be concerned only with systems for which the channel set is bi-directional, we define what it means for processors to be *neighbours* for bi-directional channel sets only.

Definition 4. If $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{I}$ is a bi-directional channel set, and (i, j) (and hence (j, i) also) is in $\mathcal{C}$, then i and j are *neighbours* in the processor network described by $\mathcal{C}$.

2.1.2 Events

With each processor in a system we associate a set of *events.* Events describe single indivisible local actions of a processor. Our model contains only three kinds of indivisible operations: *send* events, in which a single message is sent by a processor to a single destination; *receive* events, in which a single message is received by a processor; and *internal* events, which do not involve inter-processor communication. Earlier work by Taylor [Tay89] in the area of consistent-cut protocols considers an indivisible operation, called *atomic receive-send*, for receiving a message and sending multiple responses. Some of the results of this work were shown in [Cri89] to depend on the presence of this construct, and it was dropped in later work by Critchlow and Taylor [CT90a], and in this work also. We discuss atomic receive-send further in Chapter 4 when we define inhibition.

Definition 5. Given a set of messages Msg, an *event set* for a processor $i \in \mathcal{I}$ is a subset of the set $Send_i \cup Receive_i \cup Internal_i$, where

$$Send_i \equiv \{send\} \times \{i\} \times \mathcal{I} \times Msg,$$

$Receive_i \equiv \{receive\} \times \{i\} \times \mathcal{I} \times Msg$, and

$Internal_i \equiv \{internal\} \times \{i\} \times Msg.$

The event set of processor i is designated E_i; any element of E_i is an *event* of processor i.

The send event $send(i, j, m)$ occurs at processor i; i sends the single message m to the single destination processor j. The reception of the single message m from processor j at processor i is modeled by the occurrence of the event $receive(i, j, m)$ at processor i. (We assume that messages in the system are unique, so that there is no ambiguity as to which send event corresponds with a given receive event.) An internal event of a processor does not involve communication with any other processor. The event set of a processor i reflects the physical configuration of the system in that it will contain events of the form $send(i, j, m)$ only if $(i, j) \in \mathcal{C}$, and $receive(i, j, m)$ only if $(j, i) \in \mathcal{C}$.

Events describe single local actions of a processor. We now turn our attention to sequences of events occurring at a particular processor in the course of the system's functioning. We describe the *state* of a processor in terms of these sequences.

Definition 6. Let E_i be the event set of processor i. A sequence, finite or infinite, of events from E_i is called a *local history* of processor i. Any such finite sequence is also called a *local state* of i.

Definition 7. $States(E_i)$ is the set of all possible local states consisting of events from E_i. The empty sequence of events, λ, is included in $States(E_i)$.

2.2 Enabling Relations

We wish to incorporate into our model some way to describe formally the set of events that *could* occur at any given point in a processor's execution. We do this by means of a formalism known as an *enabling relation.* Conceivably, there are circumstances under which certain events cannot or should not occur at a given processor. An enabling relation describes which events, exactly, may occur next at any given point in a processor's execution. This formalism will allow us to examine the exact effect of a protocol on a system's execution: we compare the enabling relations of the original system with those of a system and protocol together.

Definition 8. An *enabling relation* M_i for a processor i with event set E_i is a subset of $States(E_i) \times E_i$. If the pair $(l_i, e_i) \in M_i$, then we say that local state l_i *enables* the event e_i.

Thus the set of events $\{e_i : (l_i, e_i) \in M_i\}$ describes which events could occur next at processor i if i is in state l_i. Note that this set may have more than one element; in fact, it may be infinite. It may also, in some cases, be empty.

We have defined $States(E_i)$ to be the set of all local states of a processor i having event set E_i. However, there may well be many states in $States(E_i)$ in which processor i will never find itself, in any execution of the system. Since i's enabling relation describes which events could happen at any point in i's execution, then any state l_i in which i actually finds itself during the course of its execution must have the property that each event of l_i is enabled by the preceding prefix of l_i. We call a local state with this property a *feasible* local state, and, by extension, any local history with the same property a *feasible* local history.

Definition 9. A local state or local history l_i of processor i with enabling relation M_i is *feasible* if for each prefix $l'_i \cdot e_i$ of l_i, l'_i enables e_i, i.e. $(l'_i, e_i) \in M_i$.

In other words, local state $l_i \equiv e_i^1 \cdot e_i^2 \cdot \ldots \cdot e_i^n$ is feasible if $(\lambda, e_i^1) \in M_i$, $(e_i^1, e_i^2) \in M_i$, …, $(e_i^1 \cdot e_i^2 \cdot \ldots \cdot e_i^{n-1}, e_i^n) \in M_i$. At each point, the next event is a "legal step", and the result is a local state which could actually occur.

We are interested in systems with two restrictions on the manner in which receive events are or are not able to occur. At any point in its execution, processor i may be willing to receive messages from one processor but not from a second processor. However, processors cannot "peek" at messages prior to having received them; they only gain knowledge of a message's contents through the appropriate receive event. Therefore i may not selectively receive messages from a single processor where the selection is based on message contents. We encode this in our formal description by placing the following restriction on enabling relations.

Property 1. For all feasible local states l_i of processor i, if $(l_i, \mathit{receive}(i,j,m)) \in M_i$, then $(l_i, \mathit{receive}(i,j,m')) \in M_i$ for any message m' from j.

Consequences of the alternative assumption are discussed at the end of Chapter 4.

The second system property that we wish to encode in the enabling relation also deals with the reception of messages. It should be the case that any message which is sent to a processor may always eventually be received by that processor. That is to say, a processor cannot "miss its chance" to receive a message which has been delivered by the communication medium. A system in which this property did not hold would in effect be inherently unreliable: a message that is delivered by the communication medium but cannot be received by the processor is in effect a message that has been lost. It would be impossible, for example, to achieve any kind of global co-operation in a system where some processor is never willing to receive messages from any of its neighbours. The following two properties of enabling relations will guarantee that messages can always be received eventually.

Property 2 states that a processor cannot reach a state in which it can no longer perform any send or internal events, and yet is unwilling to receive messages from some neighbour. If such a state were reached, than any of that neighbours messages still or subsequently in transit to that processor could not be received.

Property 2. For all feasible local states l_i of processor i such that no send or internal events are enabled, all receive events $receive(i, j, m)$ in E_i are enabled.

In any infinite execution of a processor, we wish to guarantee that there are infinitely many points during the course of the execution at which messages from a given neighbour could be received.

Property 3. In any infinite feasible local history l_i of processor i, for each neighbour j of i there are infinitely many prefixes of l_i which enable the reception of messages from j.

Note that these three properties are necessitated by our assumption that a processor does have the power to refuse to receive messages at given points in the course of its execution: receive events are not assumed to be always enabled. This is in keeping with such concurrent programming languages as Linda [Gel85] and PLITS [Fel79], but differs from such models as those of [LT89,Dil88,Jos91], in which any of a processor's input events can occur at any time. They take the view that a processor cannot necessarily control whether or not other processors send it messages, and hence whether or not there will be a message arriving on some channel at any given point in its execution. One can contemplate splitting our *receive* events into two separate events: the *arrive* event, in which a message actually is received at a processor, and the *read* event in which an arrived message is processed. All arrive events would be enabled by any state. Thus, the reception of a message would be two separate actions, the occurrence of only one of which is under local control. However, potential causality (see Definitions 10 and 11 of Section 2.3) would be transmitted through the reading of a message rather than through its arrival at a processor: until it is actually read, no information about the state of the sender can be gained. The result is that we would have added arrive events only to more or less ignore them completely: we would still need Properties (1) through (3) above to hold (of read events); the causal consistency or inconsistency of a cut (see Definition 18 of Section 2.5) would depend on read events only; the cut produced by a consistent-cut protocol (see Definition 22 of Chapter 3) would have to be distinguishable only modulo the occurrence of arrive events, etc. We therefore choose the simpler model, and only remark that a simple transformation would convert any impossibility proof in our model into a proof of a corresponding theorem in the receptive model with separate arrive and read events.

2.3 Causality Relations

The final component of our system description is a set of executions, or runs, which we associate with each system. In order to describe possible system executions, however, we require some means of providing temporal structure to sets of local histories. In synchronous systems it is possible to describe the relative order in which events at different processors take place by referring to the global system clock. In asynchronous systems, however, we do not have this option. It is possible to determine the order in which events occur at a single processor, since these processors are assumed to be sequential. Also, if a processor receives a message from one of its neighbours, then the corresponding send event must have occurred earlier at the sending processor. The fact that a certain event precedes other events implies that the first could have a causal effect on the later events. Hence this is referred to as *potential causality*, or simply causality. If two events are not related in one of these ways, or by a chain of such relationships, then there is no way to determine in which order they actually occurred. However, it is impossible, in this case, that one has any influence on the occurrence of the other. Thus potential causality gives us a partial rather than a total ordering on the events in the run.

Definition 10. Given an N-vector $r = (r_1, \ldots, r_N)$, where r_i is a local history of processor i for each i, event e *happens-immediately-before* event e', written $e \mapsto e'$, if either

1. e and e' are both events in r_i for some i, and e occurs immediately before e' in r_i, or

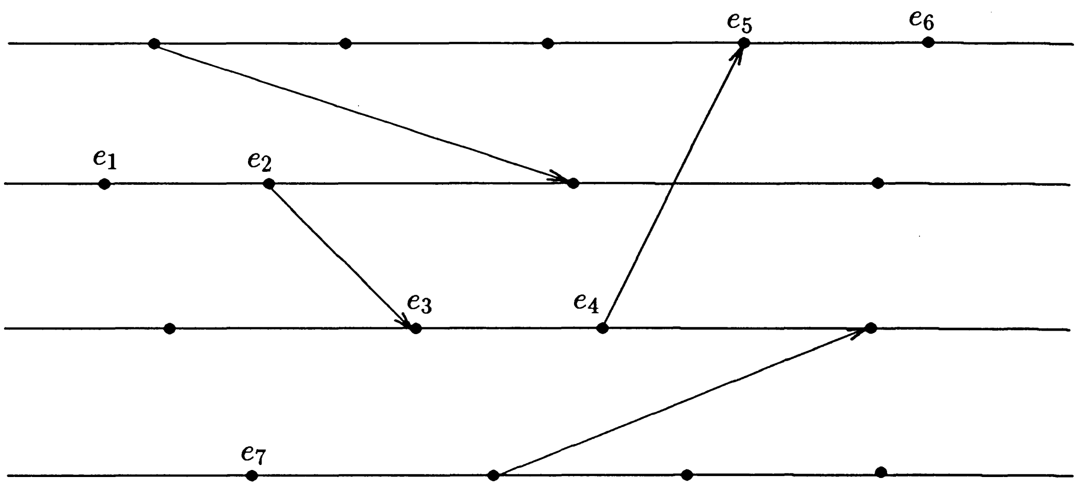


Figure 2.1: Events and the happens-before relation.

2. e is *send*(i, j, m) and e' is *receive*(j, i, m) for some processors i and j, and some message m.

Definition 11. The relation *happens-before*, denoted $\rightarrow$, is the transitive closure of happens-immediately-before: $e \rightarrow e'$ in run r if there exist events $e_1, e_2, \ldots, e_n$ such that $e \mapsto e_1 \mapsto \ldots \mapsto e_n \mapsto e'$.

Equivalently, $e \rightarrow e'$ if and only if there is an event e'' such that $e \mapsto e''$ and $e'' \rightarrow e'$.

Note that the happens-before relation holds exactly when we can be sure that one event has temporally preceded a second.

Figure 2.1 illustrates the happens-before relation. We depict an execution of a system by drawing an event line corresponding to each processor. Time moves from left to right along each line. Individual events are represented by dots on the event lines. The send and receive of a message are connected by a solid arrow. In this figure, $e_1 \rightarrow e_6$ since $e_1 \mapsto e_2 \mapsto e_3 \mapsto e_4 \mapsto e_5 \mapsto e_6$. It is not the case that $e_3 \rightarrow e_7$, nor that $e_7 \rightarrow e_3$.

We can now associate a set of *runs* with each distributed system, corresponding to the possible executions of that system.

Definition 12. Let $\mathcal{I} = \{1, \ldots, N\}$ be the processor identifier set of a distributed system; E_i be the event set of processor i for each $i \in \mathcal{I}$; and M_i be an enabling relation for each $i \in \mathcal{I}$. A vector r of local histories is a *run* of this system if the following conditions hold.

1. For all $i \in \mathcal{I}$, r_i is a feasible local history of processor i.
2. The reflexive closure of happens-before is a partial order on the set of events included in r, i.e. there are no two events e and e' such that $e \rightarrow e'$ and $e' \rightarrow e$ in r.
3. For every local history r_i, if r_i is finite then there is no send or internal event e_i such that $(r_i, e_i) \in M_i$.

The first condition requires that each component of r represents a legal sequence of steps as laid out by the enabling relation of the appropriate processor. The second rules out the possibility of one event happening-before a second, which in turn happens-before the first; since the happens-before relation is a subset of temporal precedence, this would make no sense physically. This final condition states that no processor will arbitrarily cease execution when it could yet proceed with some send or internal event. We discuss the question of receive events in the next section.

2.4 System Definition

Using the elements described thus far, we can now formally define a system.

Definition 13. A *system* $\mathcal{S}$ is a 5-tuple $(\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$, where

1. $\mathcal{I} = \{1, ..., N\}$ is a set of processor identifiers;
2. $\mathcal{C}$ is a bi-directional connected channel set for $\mathcal{I}$;
3. $\mathcal{E} = (E_1, E_2, \ldots, E_N)$ is an N-vector of event sets where for each i, E_i contains events of the form $send(i, j, m)$ or $receive(i, j, m)$ only if $(i, j) \in \mathcal{C}$;
4. $\mathcal{M} = (M_1, M_2, \ldots, M_N)$ is an N-vector such that for each $i \in \mathcal{I}$, M_i is an enabling relation on the set of events E_i, satisfying Properties 1 through 3; and
5. $\mathcal{R}$ is the set of all runs over $\mathcal{I}, \mathcal{E}$, and $\mathcal{M}$.

Throughout this thesis we confine our interest to systems in which both the processors and the communication medium are reliable. Processors cannot fail; any message that is sent is eventually received at the correct destination; any message that is received by a processor has actually been sent to that processor by another; and no message is altered in transmission. Note that our Property 2 of enabling relations does not permit crash failures: there cannot be an internal "crash" event after which no further messages can be received. However, though we consider only systems in which Properties 2 and 3 hold of enabling relations, it is entirely possible that there are vectors of local histories which satisfy the definition of run, but which do not exploit these properties; that is, runs in which not every message that is sent is also received. We do not wish at this point to include these in our formal system description for the same reason as that given above: we are interested in reliable system behaviour, and an unreceived message is indistinguishable from a lost one. We therefore have the following definition.

Definition 14. A run r is said to be *completed* if for every pair of processors $i, j \in \mathcal{I}$, if the event $send(i, j, m)$ occurs in the local history r_i, then $receive(j, i, m)$ appears in r_j.

Note that this condition, together with clause 3 in the definition of run, implies that if r_i is finite in run r, then the only events that are enabled by r_i are receive events $receive(i, j, m)$ such that either (a) $receive(i, j, m)$ already appears in r_i, or (b) $send(j, i, m)$ does not appear in r_j.

The completeness condition on runs guarantees that all messages that are sent are eventually received at the correct destination. We need to ensure, however, that no messages are spawned or altered by the communication medium. This is formalized by the final restriction in the following definition.

Definition 15. A system $S = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ is *reliable* if $\mathcal{R}$ is the set of all *completed* runs over $\mathcal{I}$, $\mathcal{E}$, and $\mathcal{M}$, and for each $r \in \mathcal{R}$, r contains no event $receive(i, j, m)$ for which there is not a corresponding send event $send(j, i, m)$ in the appropriate local history.

Throughout the remainder of this thesis, the systems under discussion are assumed to be reliable.

We will sometimes be concerned with modeling systems whose channels exhibit *first-in-first-out* behaviour.

Definition 16. A system $S = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ is a *FIFO* system, if for all $r \in \mathcal{R}$ and all neighbouring processors i and j in $\mathcal{I}$, whenever $send(i,j,m) \rightarrow send(i,j,m')$ in r, then $receive(j,i,m) \rightarrow receive(j,i,m')$ in r.

This definition requires that messages sent along a particular channel arrive at their destination in the same order as they were sent. It does not require that all communication between processors be completed in the same order as it was initiated. For instance, suppose that in a run of FIFO system S, processor i sends a message m_1 directly to processor j, and then sends a message m_2 to a third processor k. If k then "forwards" m_2 to j in the form of the message m_3, it is possible that m_3 be received by j before m_1 is.

2.5 Partial Runs and Consistent Cuts

A run is a description of an entire (possibly infinite) execution of a system. We wish now to describe what the system might look like at some specific, finite point in the course of its execution.

For each processor a local state will describe the sequence of steps taken; there will be no infinite local histories. What must be true of the vector of local states to make it meaningful as an instantaneous picture of the global system state? Clearly, the first two conditions in the definition of a run—that each local state be feasible, and that there be no circular causality—must hold, for the same reasons as those given earlier. Equally clearly, the third condition—that there be no further send or internal events enabled—need not hold. Since the system in question is assumed to be reliable, we need an additional constraint: that there be no receive event for which the corresponding send event has not yet occurred. (If the system is unreliable, then a send-less receive is a possibility at any time: the message being received may have been generated by the communication network.) Also, if the system in question is FIFO, then the FIFO property must hold of the vector of local states in order for it to correspond to a possible real-time configuration of the system.

Definition 17. Let $S = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ be a system, and $r' = (r'_1, \ldots, r'_N)$ be a vector of local states. We call r' a *partial run* of S if the following conditions hold:

1. for all $i \in \mathcal{I}$, r_i is a feasible local history of processor i; and
2. the reflexive closure of happens-before is a partial order on the set of events included in r.
3. for each receive event in r there must be a corresponding send event in the appropriate local history.

If S is a FIFO system and $send(i,j,m) \rightarrow send(i,j,m')$ in r, then either (a) neither of the corresponding receives appears in r, (b) $receive(j,i,m)$ only appears in r, or (c) $receive(j,i,m) \rightarrow receive(j,i,m')$ in r.

The definition of partial run gives the minimum conditions that must be true in order for a vector of local states to represent a global state of a system. We will show in Chapter 5 that the name "partial run" is justified in the sense that any partial run can be extended to a completed run.

Definition 18. Given a partial run r' of a system, for any run r such that r'_i is a prefix of local history r_i for all i, r' is called a *consistent cut* of r. Equivalently, given a total run r and an arbitrary vector $(l_1, l_2, \ldots, l_N)$ of prefixes of r,

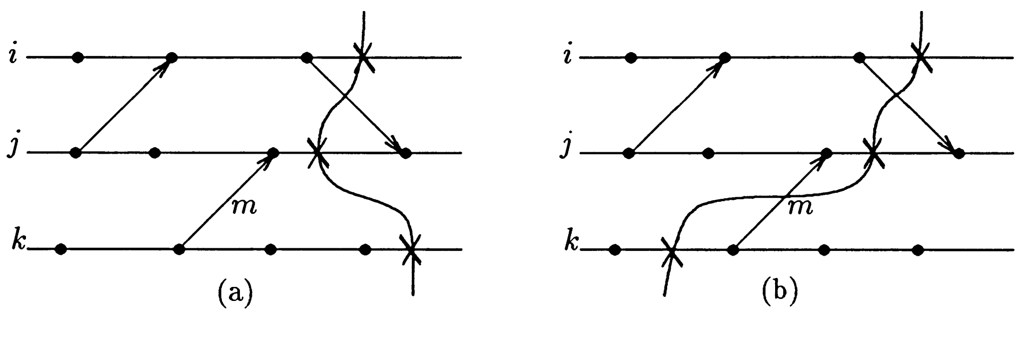


Figure 2.2: Consistent (a) and inconsistent (b) cuts in a run.

$(l_1, l_2, \ldots, l_N)$ is a consistent cut of r if and only if it does not include the reception of a message whose send is not also included.

Note that a consistent cut has the following property: if an event e is contained in the cut, then any other event e' in the run such that $e' \rightarrow e$ is also included in the cut.

Figure 2.2 illustrates consistent and inconsistent cuts in a run. The states to the left of the "X"'s in part (a) form a consistent cut, since for every receive event contained in them, the corresponding send event also appears. In part (b), however, the event *receive*(j, k, m) is included in the indicated cut, while the event *send*(k, j, m) is not. We refer to a message such as m as an inconsistent message.

We have defined partial runs and consistent cuts only in the context of systems which are assumed to be reliable. If we were to include in our study systems where communication is unreliable, we could relax clause 3 in the definition of partial run: as noted above, a send-less receive is a possibility at any time in such a system. However, in this case, a partial run will be a consistent cut in an extending run only if the sends corresponding to these send-less receives do not appear in the extension.

2.6 Summary

In this chapter, we have presented a formal model for asynchronous distributed systems. This model includes the notion of an enabling relation, which provides an exact description of what events could occur at any point in a processor's execution. We have discussed the properties that should hold of the various system components in order for them to describe the properties of the particular systems in which we are interested. We have also introduced potential causality as a method for imposing order on the events in a system execution, and the related notion of consistency.

Chapter 3

Consistent-Cut Protocols

In this chapter we give a formal definition of a *protocol* for a distributed system. Informally, a protocol specifies for each processor a set of protocol events, and an algorithm to determine the manner in which that processor should execute the events so as to accomplish the goal of the protocol. More formally, a protocol maps an original system to a new one by introducing new events, new enabling relations on sequences of system and protocol events, and a new set of associated runs. (The protocol does not, of course, change the physical configuration of the system.)

We have chosen a model in which system and protocol events are strictly separate. There can be no "hybrid" events such as the merging of system and protocol messages; this type of merging occurs in *piggybacking* techniques, in which protocol information is added to the contents of system messages. We discuss piggybacking further in Chapter 9. Here, the distinction between system and protocol events will allow us to define *inhibition*, as discussed in Chapter 4

We introduced in Chapter 2 the notion of a consistent cut, and the correspondence of the consistent cuts in a run to possible global states of the system in the course of its execution. In the second part of this chapter we define *consistent-cut protocols* or *CCPs*: these are protocols which in every protocol run will designate a set of local states, one per processor, in such a way that these states together form a consistent cut. Consistent-cut protocols have numerous distributed applications such as system checkpointing [Rus77,CL85,KT87], deadlock detection [BT87], distributed termination [Fra80], and broadcasting [BJ87]. CCPs differ from *snapshot* [CL85,KT87] protocols in that we are not concerned with recording the states of channels. (Of course, impossibility results for consistent-cut protocols directly apply to protocols, such as snapshots, that perform other tasks in addition to determining consistent cuts.)

We also present here three well-known consistent-cut protocols, and investigate the circumstances under which each protocol is successful. We consider whether (1) the protocol works for non-FIFO systems or only FIFO systems, and (2) all messages or only system messages must be consistent with respect to the cut designated by the protocol. We present proofs that the protocols are correct under the circumstances claimed.

3.1 Protocols

A protocol is, in effect, a function from systems to systems. Introducing a protocol into a distributed system produces a new system with identical identifier and channel sets: a protocol does not change the physical configuration of a system. The remaining three components of our formal system definition will, however, change. We examine these changes and introduce some terminology before proceeding to the actual definition of a protocol.

Firstly, none of the events of the original system can be lost in the transformation by the protocol. New events, however, may be introduced. We distinguish these as *system* and *protocol* events.

Definition 19. Let $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ be a system and $\mathcal{P}$ a protocol mapping $\mathcal{S}$ to $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}', \mathcal{R}')$. For each processor $i \in \mathcal{I}$, an event e_i such that $e_i \in E_i$ is a *system event*, while an event e_i such that $e_i \in E_i'$ but $e_i \notin E_i$ is a *protocol event.*

If the event *send*(i, j, m) is a system (resp. protocol) event, then we also refer to the message m as a system (resp. protocol) message.

Given a sequence of events occurring at a processor during an execution of the new system, we often wish to separate out the subsequence of system events. For this we have the following definition.

Definition 20. Given a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ and its image under protocol $\mathcal{P}$, $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}', \mathcal{R}')$, the function *SysEvents* maps a sequence of events from $E_1' \cup \ldots \cup E_N'$ to its projection onto the set of events $E_1 \cup \ldots \cup E_N$.

The enabling relation for each processor i in $\mathcal{P}(\mathcal{S})$ is a subset of $States(E_i') \times E_i'$. However, it should be the case that in the absence of protocol events, the enabling relation is as in the original system. That is, until the execution of the task designated by the protocol has actually commenced, the system functions entirely as it did originally.

Finally, we require that the presence of the protocol does not prevent the underlying system from performing its own task. For any run $(r_1', \ldots r_N')$ of the new system, the run $(SysEvents(r'_1) \ldots SysEvents(r'_N))$ should be a valid behaviour of the original system.

We now give our formal definition of a protocol.

Definition 21. A *protocol* $\mathcal{P}$ is a function which maps a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ to a new system $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}', \mathcal{R}')$ such that

1. if $\mathcal{E} = (E_1, \ldots E_N)$ and $\mathcal{E}' = (E'_1, \ldots E'_N)$, then $E_i \subseteq E'_i$ for each $i \in \mathcal{I}$;

2. for all processors i, all $l_i \in States(E_i)$, and all $e_i \in E_i$, $(l_i, e_i) \in M_i$ if and only if $(l_i, e_i) \in M'_i$;

3. for all runs $r' = (r'_1, ..., r'_N)$ in $\mathcal{R}'$, the run $(SysEvents(r'_1), ..., SysEvents(r'_N))$ is in $\mathcal{R}$; and

4. if $\mathcal{S}$ satisfies the FIFO condition then so does $\mathcal{P}(\mathcal{S})$.

Note that because $\mathcal{P}(\mathcal{S})$ is a system, it will automatically satisfy conditions 1 through 5 of Definition 13 in Chapter 2, as well as the reliability conditions of Definition 15. We refer to runs of this modified system as *runs of the protocol* with respect to the original system. Also note that the third requirement in the definition above is not sufficient to guarantee that the FIFO property is not lost under a protocol $\mathcal{P}$: it governs only system messages. To ensure that protocol messages also obey the FIFO condition, both with respect to system messages and to each other, we add the fourth clause above. Finally, we remark that the definition leaves open the possibility that some valid system behaviours may be lost: it is not guaranteed that for any given run r of the original system there will be a protocol run $r' = (r'_1, \ldots, r'_N)$ such that $r = (SysEvents(r'_1), \ldots, SysEvents(r'_N))$. The protocols which we present, however, do have this stronger property; and our impossibility results using the weaker definition would of course also hold if we required all CCPs to have it.

3.2 Consistent-Cut Protocols

We now turn our attention to a specific class of protocols for distributed systems: namely, *consistent-cut protocols.*

Definition 22. A *consistent-cut protocol (CCP)* is a protocol $\mathcal{P}$ which, for every system $\mathcal{S}$ and every protocol run r in $\mathcal{P}(\mathcal{S})$, will designate a set of local states $cut(r)$ called *cut states* in such a way that:

1. for all r, $cut(r)$ is a consistent cut of r;
2. for any two protocol runs r and r' both containing local state l_i of processor i, if l_i is i's cut state in run r then it is also i's cut state in run r';
3. for any partial run r of the original system $\mathcal{S}$, there is a protocol run r' of $\mathcal{P}(\mathcal{S})$ that extends r; i.e. for each i, r_i is a prefix of r'_i; and
4. for every run r of the protocol and every processor i, there is a prefix l_i of local history r_i such that every protocol event which occurs in r_i is contained in l_i, and for every extension l'_i of l_i, $(l'_i, e_i) \in M_i$ implies that $(l'_i, e_i) \in M'_i$.

The second condition implies that the cut state of each processor is distinguishable to that processor; in other words, each processor i "knows" that any messages sent subsequently to j will not be received in j's cut state, regardless of the sequence of steps that j may have taken in the meantime. More formally, each processor achieves concurrent common knowledge [PT88] of the fact that it has reached the cut designated by the protocol. The third condition implies that a CCP cut may occur anywhere in the run of the underlying system; this eliminates CCPs that, for example, only distinguish the cut $(\lambda, ..., \lambda)$ at the beginning of a run. The final condition states that the action of the protocol must be finite.

We sometimes consider CCPs in which $cut(r)$ may contain inconsistent *protocol* messages; this implies that condition (1) is changed to require that $(SysEvents(l_1), \ldots, SysEvents(l_N))$ be consistent.

3.3 Examples of CCPs

In this section we give three CCPs and the proofs that they perform as claimed. The first two are *flooding* algorithms, in which messages are sent along every channel in the system [Cha82,CL85]. *Flood1* is essentially the Chandy and Lamport checkpointing algorithm [CL85]. *Flood2* is similar to that of [Tay89], with the indivisible receive-send mechanism replaced by local receive inhibition. The third algorithm, *Tree*, is a two-phase spanning tree protocol from [Tay89]; it is similar to protocols in [Fra80,Awe85].

In both $Flood1$ and $Flood2$, messages are sent along every channel in the system, beginning with messages sent by a distinguished initiator I. The processors reach their cut states either immediately upon, or closely following, the first reception of a protocol message. A consistent cut essentially occurs because any system message sent after the protocol messages must arrive after the cut due to FIFO channels. Additionally, either the sending or receiving of system messages must be suspended during the interval in which protocol messages are sent. In $Flood1$, in which the cut is reached at the beginning of that interval, system mes-

- Initiator I, at any time :

  start CCP; **(Cut)** *send*(I, j, Cut) to each neighbour j before sending further system messages to j.

- Other processors i, immediately upon the first occurrence of a receive event of the form *receive*(i, k, Cut) :

  (Cut) *send*(i, j, Cut) to each neighbour $j \neq k$ before sending further system messages to j.

Figure 3.1: **Flood1** (FIFO channels, system messages consistent)

sages sent to a neighbour after the cut and prior to the protocol message could be inconsistent and are therefore not permitted by the protocol. In $Flood2$, the cut is reached at the end of the interval. Consequently, it is the reception of system messages during this interval which could cause inconsistency and must be prevented. This phenomenon is called *inhibition*; we will give a formal definition and a detailed examination of inhibition in Chapter 4.

Each **(Cut)** in Figure 3.1 indicates a state in the consistent cut. We assume that an internal protocol event, denoted *Start CCP*, begins each initiator's protocol execution.

Note that conditions (2) through (4) in the definition of consistent-cut protocols are certainly met by $Flood1$: for each processor, the first protocol event that occurs in its local history will be the last event of the cut state, and so the cut is distinguishable; the *Start CCP* event, and hence all further protocol events can occur arbitrarily late in the run; and there are only finitely many protocol events, and finite intervals over which events are delayed. We need only show that the designated states together form a consistent cut with respect to system messages. (There will be inconsistent protocol messages in any run of $Flood1$: they are all sent out after the sending processor has reached its cut state, but no processor (other than the initiator) reaches its cut state before receiving one.)

Proof of correctness of protocol *Flood1*: Suppose there is some run r of protocol $Food1$ in which the system send event *send*(i, j, m) occurs after i's cut state in the run, but *receive*(j, i, m) appears in j's cut state.

Case 1: Suppose that i does not reach its cut state upon receiving a Cut message from j; i.e. either i is the initiator or i reaches its cut state upon receiving a Cut message from some processor other than j. In this case, i will send a Cut message to j after its (i's) cut state, and before sending any post-cut system messages to j. Thus i sends the *cut* message before it sends the inconsistent message m. Because the system is FIFO, the event *receive*(j, i, Cut) will occur at j before the event *receive*(j, i, m). (See Figure 3.2, part (a). The solid "X" marks i's cut state, and the dotted "X"'s the possible locations of j's cut state.) However, j will reach its cut state upon receiving the Cut message from i if it has not already done so. Therefore it is impossible that the event *receive*(j, i, m) be included in j's cut state, contradicting our original assumption. Thus, if there is an inconsistent system message from i to j, it must be that i reaches its cut state upon receiving a Cut message from j. Let us consider this case. Case 2: Suppose i reaches its cut state upon receiving a Cut message from j. The event *receive*(i, j, Cut) will appear in processor i's cut state, while the event *send*(j, i, Cut) occurs after j's cut state. However, we have assumed

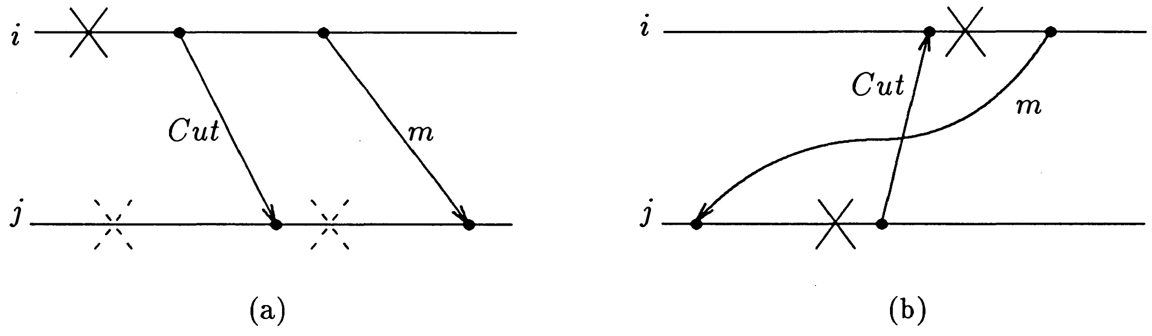


Figure 3.2: Proof of correctness of $Flood1$. Part (a) illustrates Case 1 in the proof; part (b) illustrates Case 2.

that $send(i,j,m)$ occurs after i's cut state, and therefore after $receive(i,j,Cut)$. Similarly, $receive(j,i,m)$ occurs in j's cut state, and thus before $send(j,i,Cut)$. (See Figure 3.2, part (b).) We now have

$$send(j,i,Cut) \rightarrow receive(i,j,Cut) \rightarrow send(i,j,m) \rightarrow$$

$$receive(j,i,m) \rightarrow send(j,i,Cut).$$

But there cannot be circular causality in a run, and we have reached a contradiction. In sum, there cannot be an inconsistent system message in any protocol run. ∎

The protocol $Flood2$ is described in Figure 3.3. Once again, it is easy to see that conditions (2) through (4) in the definition of CCP are satisfied by $Flood2$. We must still show that the designated states form a consistent-cut, this time with respect to all messages.

Proof of correctness of protocol *Flood2*: Suppose that there is an inconsis-

- Initiator I, at any time :

  start CCP; $send(I, j, Cut)$ to each neighbour j before receiving further messages. (**Cut**)

- Other processors i, immediately upon the first occurrence of an event of the form $receive(i, k, Cut)$:

  $send(i, j, Cut)$ to all neighbours $j \neq k$ before receiving further messages. (**Cut**)

Figure 3.3: **Flood2** (FIFO channels, all messages consistent)

tent message m from processor i to processor j in some run r of protocol $Flood2$. If i has sent j a Cut message in run r, then that message will have been received at j before the inconsistent message m, since i sends all Cut messages before reaching its cut state, and the system is FIFO. However, j does not receive any messages from i after the event $receive(j, i, Cut)$ and before reaching its cut state. Therefore the event $receive(j, i, m)$ would occur after j's cut state. Since m is an inconsistent message by assumption, we conclude that i does not send a Cut message to j in the run.

It must be the case, then, that the Cut message from j was the first Cut message to be received at i in the run r. Therefore the event $receive(i, j, Cut)$ occurs in i's cut state. Since $send(i, j, m)$ occurs after i's cut state, we have $receive(i, j, Cut) \rightarrow send(i, j, m)$. But it must also be the case that $receive(j, i, m) \rightarrow send(j, i, Cut)$, since there are no messages from i received by j between $send(j, i, Cut)$

and the end of j's cut state. Now, however, we have

$$send(j, i, Cut) \rightarrow receive(i, j, Cut) \rightarrow send(i, j, m) \rightarrow$$

$$receive(j, i, m) \rightarrow send(i, j, Cut).$$

Since there cannot be circular causality, we must conclude that there can be no inconsistent message. ∎

The flooding protocols are correct for FIFO systems because the presence of the protocol messages between each pair of neighbours forms a barrier against inconsistency for messages sent afterwards. If the system is not FIFO, however, these protocol messages cannot perform this service, since system messages sent afterwards may arrive at the destination processor first. We therefore need a different approach for non-FIFO systems. Protocol *Tree* uses a three-phase method. A spanning tree of the network, assumed to be known in advance, is used to minimize communication. Again there is a distinguished initiator I (the root of the spanning tree). Three messages, *PrepareCut, Cut,* and *Resume* are sent respectively down, up, and back down the spanning tree. System send events are suspended as *Cut* messages move up the tree, and the suspension is lifted as *Resume* messages move down the tree. In sum, any inconsistent message sent after the cut—and after the re-starting of system send events—would have to be received before being sent, due to causal chains from the receiver to the initiator and from the initiator to the sender.

Protocol *Tree* is described in Figure 3.4, and illustrated in Figure 3.5. Conditions (2) through (4) being satisfied, we must show that the designated states form a consistent cut with respect to all messages.

Let T be a spanning tree of the network rooted at I.

Let $parent(i)$ and $children(i)$ be the parent and children of i in T.

(1) Initiator I: start CCP; send $PrepareCut$ to $children(I)$.

(2) Each internal node i, after receiving $PrepareCut$: send $PrepareCut$

to $children(i)$.

(3) Leaf nodes i, after receiving $PrepareCut$: Disable system sends; send Cut to $parent(i)$. **(Cut)**

(4) Each internal node i, after receiving Cut from $children(i)$: disable system sends; send Cut to $parent(i)$. **(Cut)**

(5) Initiator I, after receiving Cut from $children(i)$: **(Cut)** send $Resume$ to $children(I)$.

(6) Internal nodes i, after receiving $Resume$ from $parent(i)$: send $Resume$ to $children(i)$; enable disabled system sends.

(7) Leaf nodes i, after receiving $Resume$ from $parent(i)$: enable disabled system sends.

Figure 3.4: **Tree** (non-FIFO channels, all messages consistent)

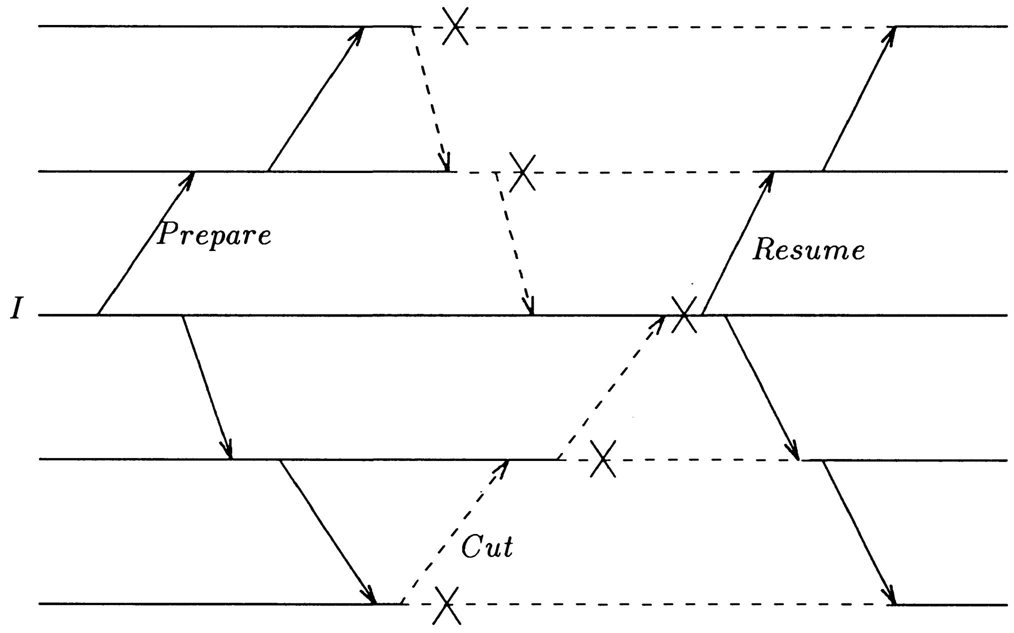


Figure 3.5: The three phases of protocol messages sent in a run of the protocol *Tree*. Dashed intervals contain no system send events.

Proof of correctness of protocol *Tree*: Suppose that there is a run r of $Tree$ in which there is an inconsistent message m from processor i to processor j. In order to reach a contradiction, we wish to prove the following claim: for any processors p and q, the last event e_p of p's cut state must happen-before the first event e'_q after q's cut state which could be a system send to p. If we can prove this, then we have a contradiction, because it will then be the case that for inconsistent message m,

$$e_j \rightarrow e'_i \rightarrow send(i,j,m) \rightarrow receive(j,i,m) \rightarrow e_j$$

and there is causal circularity.

Let p be an arbitrary processor. The last event of p's cut state for $p \not\equiv I$ is $send(p, parent(p), Cut)$. (See Figure 3.5.) Each processor waits to receive Cut messages from all of its children and then relays the Cut message to its parent. Since I is at the root of the spanning tree T, this means that

$$send(p, parent(p), Cut) \rightarrow receive(I, p^*, Cut)$$

where p^* is the last of I's children from which it receives a Cut message in run r. This is still an event of I's cut state (in fact, it is the last event), and thus happens-before I's first post-cut system send. So far, then, we have proved our claim for $p \not\equiv I$ and $q \equiv I$.

Now, I does not send *Resume* messages until it has received Cut messages from all of its children. No processor $q \not\equiv I$ sends a *Resume* message until it receives one from its parent, nor can it send system messages before this point. Again, since I is the root of the spanning tree, we have that

$$receive(I, p^*, Cut) \rightarrow receive(q, parent(q), Resume)$$

and the claim holds for $p \equiv I$ and $q \not\equiv I$.

Combining the two results above, we have

$$send(p, parent(p), Cut) \rightarrow receive(I, p^*, Cut) \rightarrow receive(q, parent(q), Resume)$$

and our claim holds in the final case of $p, q \not\equiv I$. ∎

3.4 Summary

In this chapter we have given formal definitions of protocols and of consistent-cut protocols. We have presented three examples of consistent-cut protocols that are correct under various system and protocol assumptions; we have discussed these circumstances, and presented proofs of correctness. In discussing these protocols we have indicated that, in each, some subset of system events must be suspended by the protocol in order for it to succeed. We now proceed to Chapter 4 where this phenomenon is formally defined as *inhibition*, and examined with more care.

Chapter 4

The Inhibition Spectrum

We wish to investigate the degree to which a consistent-cut protocol must interfere with an underlying system in order to determine causally-consistent global states in the course of an execution. The definitions of this section will allow us to characterize nine different categories of protocols with respect to their inhibitory characteristics.

A protocol runs on top of, or in conjunction with, the normal execution of an underlying system: system and protocol events will be interspersed at each processor, but the subsequences of system events, taken together, should form a valid run of the original system. Thus the presence of the protocol should not prevent the underlying system from performing correctly. What the protocol may do, however, is to prevent a processor from functioning normally *for some finite interval.* For example, it may force a processor to wait upon occurrence of some protocol event before proceeding with system events that would have been permitted in the absence of the protocol. We refer to this phenomenon as *inhibition*; in [BF88], this concept is termed *freezing* and is defined for a general notion of superimposed processes. We make an important distinction between *local* and *global* inhibition: that is, between cases where a protocol only delays events until some number of local actions—sends and internal events—have been performed and cases where it delays events while a processor waits for communication from another processor. The latter is in some sense a more serious form of inhibition, as one processor may have to wait upon the reception of a message from a much slower neighbour. If processors or communication medium are unreliable then the wait may be indefinite. We will prove in subsequent chapters that the character—local or global—of the inhibition that a protocol uses is a crucial factor in the existence and efficiency of protocols for determining consistent global states.

In Chapter 3 we presented three consistent-cut protocols and investigated the conditions under which they succeed. We will now add questions of inhibition to our analysis of consistent-cut protocols. Together, these issues produce thirty-six possible combinations of protocol and system properties; we re-investigate to which cases our three protocols pertain. In each case for which none of these protocols suffices, we can prove that no protocol with the appropriate properties and effects exists. These arguments are contained in Chapters 6 and 7.2; this chapter ends with a tabular summary of all existence and impossibility results.

4.1 Inhibition

First, we define inhibition essentially as in [Tay89]. We take the intermediate approach of first defining the notion of *disabling* an event. Suppose that, in an original system, a given event could occur immediately following a particular sequence of events. If, in the system created by a protocol $\mathcal{P}$, the same sequence of system events, with protocol events interspersed, does not enable that event, then we say it has been disabled (by the protocol).

Definition 23. Let $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$ be a system and $\mathcal{P}$ be a protocol with $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}', \mathcal{R}')$. An event e_i is *disabled* in state l_i of protocol run r if the subsequence of system events in l_i enables e_i in the original system, but l_i does not enable e_i:

$$(SysEvents(l_i), e_i) \in M_i$$

$$(l_i, e_i) \notin M_i'$$

Definition 24. A protocol is *non-inhibitory* if no event is disabled in any run of the protocol. Any protocol which does disable system events is *inhibitory*.

A non-inhibitory protocol, then, does not interfere with the running of the underlying system. Note that there need be only one system $\mathcal{S}$ and one run of $\mathcal{P}(\mathcal{S})$ in which events are disabled in order for the protocol $\mathcal{P}$ to be inhibitory.

4.2 Local and Global Inhibition

We distinguish two types of inhibition, *local* and *global*: recall that we wish to separate cases where a protocol delays events only until some number of local actions—send or internal events—have been performed from cases where it delays events while waiting for communication from another processor.

Definition 25. Suppose event e_i is disabled by a protocol in state l_i of run r. We say e_i is *locally delayed* if there exists a run r' such that

1. r_i' contains an extension $l_i{}'$ of l_i;

2. e_i is not disabled by the protocol in l_i' or any extension of l_i'; and

3. $l_i' - l_i$ contains no receive events.

Note that in the run r itself, there may well be receive events occurring between l_i and the point at which e_i is no longer disabled. However, this may be purely coincidental; thus provided that there is *some* run where there is no intervening receive, we can conclude that the duration of the delay is under local control. We require that e_i not be disabled at any point after l_i' to avoid the following scenario. Suppose in some run r there is an event e_i "locally" delayed in l_i, and there is a receive event occurring between l_i and the end of the disabling. There will be another run r' with an extension l_i' such that $l_i' - l_i$ contains no receive events and e_i is no longer disabled. However, e_i may subsequently be disabled again, say in l_i'', and in run r' there is a receive event between l_i'' and the end of the disabling. This could be endlessly repeated: the definition would leave open the possibility that the last disabling of e_i requires communication to terminate it.

We use the terminology that an event may be locally or globally *delayed*, whereas in doing so a protocol exhibits local or global *inhibition*. Given the definition of local delay, locally and globally inhibitory protocols are defined in a straightforward manner.

Definition 26. A protocol is *locally inhibitory* if any event disabled in any run of the protocol is locally delayed. An inhibitory protocol that does not have this property is *globally inhibitory*.

This implies that, in a globally inhibitory protocol, some events are delayed while waiting for communication from another processor.

Finally, we also consider whether or not send events or receive events are delayed by an inhibitory protocol. Recall that we will be interested in asking questions about the degree of inhibition necessary to allow a protocol to distinguish possible global states in asynchronous runs. Because the occurrence (or lack thereof) of internal events cannot effect the consistency of a set of states, the ability (or lack thereof) to inhibit internal events is not an issue.

Definition 27. A protocol exhibits *send inhibition* if some (locally or globally) delayed events are send events. Likewise, a protocol exhibits *receive inhibition* if some (locally or globally) delayed events are receive events.

Note that local send and receive inhibition together suffice to allow a protocol to create, in effect, new "atomic" actions, such as the atomic receive-send of [Tay89]: the protocol need only delay all other events until the requisite series of send and internal events has been performed.

Consider now the three consistent-cut protocols of Chapter 3: $Flood1$, $Flood2$, and $Tree$. We wish to categorize these protocols according to the type of inhibition they display. $Flood1$ does not allow a processor to send system messages for an interval following the first reception of a Cut message. $Flood2$, on the other hand, disables receive events. In both protocols, the inhibition is only local because a sequence of send events ends the disabling, with no necessary communication from another processor. Thus $Flood1$ exhibits local send inhibition, and $Flood2$ local receive inhibition. $Tree$, in common with $Flood1$, disables send events for an interval during its execution. However, in $Tree$'s case, a message chain from the initiator, ending with the event $receive(p, parent(p), Resume)$, is required to re-enable the send events of processor p. This is therefore an exam-

Name	Channels	Inhibition	Protocol Messages
Flood1	FIFO	local send	inconsistent
Flood2	FIFO	local receive	consistent
Tree	non-FIFO	global send	consistent

Figure 4.1: Protocol characteristics.

ple of global send inhibition. We summarize the properties of these protocols in Figure 4.1.

4.3 The Inhibition Spectrum

We have defined three types of inhibition based on whether send events are delayed locally, globally, or not at all, and similarly for receive events. This results in nine different combinations of protocol capabilities. For example, a protocol exhibits local receive inhibition and global send inhibition if it delays both receive and send events, and no delayed receive event requires communication to be re-enabled.

Since a globally inhibitory protocol may both globally and locally delay events, it follows that global inhibition of a particular type of event (send or receive) is at least as strong as local inhibition of that event. Clearly, local inhibition of a particular type of event is at least as strong as no inhibition of that event. Thus, for example, when other factors remain constant, global send inhibition is at least as strong as local send inhibition, which is at least as strong as no send inhibition. This is independent of the type of protocol under consideration.

In sum, we now have thirty-six combinations of system and protocol assumptions, the issues being whether:

- the protocol works for non-FIFO systems or only FIFO systems;
- all messages or only system messages must be consistent with respect to the cut designated by the protocol;
- the protocol uses no, local, or global inhibition;
- send events, receive events, or both must be delayed.

Figure 4.2 illustrates which CCPs are successful under varying system and protocol characteristics. For every case in which none of these protocols can be used, we can demonstrate impossibility as designated by IMP. Specific impossibility results for the IMP† cases are presented in the following chapters; the remaining impossibilities are immediate consequences of those results.

Recall that we do not assume that processors can choose to receive some messages but not others on a single channel; the appropriateness of this capability depends on the level of software being modeled. Also, the behavior of "selective receive" is ambiguous in the case of FIFO channels. If this mechanism is allowed in non-FIFO systems, then there is a protocol symmetric to *Tree*, in which receives are globally disabled rather than sends; it is successful regardless of send inhibition and protocol message consistency. This case is less interesting from the standpoint of understanding the precise limitations on attaining CCPs.

		System and Protocol Messages Consistent		Only System Messages Consistent	
		nonFIFO	FIFO	nonFIFO	FIFO
No Send Inhibition	No Receive Inhibition	IMP	IMP	IMP	IMP†
	Local Receive Inhibition	IMP	Flood2	IMP	Flood2
	Global Receive Inhibition	IMP	Flood2	IMP	Flood2
Local Send Inhibition	No Receive Inhibition	IMP	IMP†	IMP	Flood1
	Local Receive Inhibition	IMP	Flood2	IMP	Flood1,Flood2
	Global Receive Inhibition	IMP	Flood2	IMP†	Flood1,Flood2
Global Send Inhibition	No Receive Inhibition	Tree	Tree	Tree	Tree, Flood1
	Local Receive Inhibition	Tree	Tree, Flood2	Tree	Tree, Flood1,Flood2
	Global Receive Inhibition	Tree	Tree, Flood2	Tree	Tree, Flood1,Flood2

Figure 4.2: Consistent-cut protocols and impossibilities in the inhibition spectrum.

4.4 Summary

In this chapter, we have given a formal definition of inhibition. We have discussed the different degrees of inhibition that a protocol can display, distinguishing between local and global inhibition, send and receive inhibition. We have re-examined the three consistent-cut protocols presented in Chapter 3 to analyze their inhibitory behaviour. We have indicated to which cases in our thirty-six combinations of protocol and system assumptions these three protocols apply. We claim, and will demonstrate in the following chapters, that in all cases to which none of these protocols pertain, no consistent-cut protocol with the requisite characteristics exists.

Chapter 5

Preliminary Lemmas

In this chapter we present six lemmas that will be useful in the impossibility and complexity results of the following chapters. The first five lemmas concern the structure of the collection of runs in asynchronous systems in general. Lemma 1 examines the relationship between partial runs and completed runs; the proof is based on an argument presented in [Tay89]. Lemmas 2 and 3 demonstrate how the reception of messages can be "pushed back"; that is, a given receive event can occur almost arbitrarily early in the course of a system's execution. Lemmas 4 and 5 discuss the connection between receive events and the flow of potential causality. The proofs of the first three of these lemmas originally appeared in [Tay89]; we present them here for completeness. The final lemma of the chapter concerns the structure of the set of runs containing locally delayed events.

5.1 Partial and Completed Runs

In Chapter 2 we discussed how a vector of local states could represent a simultaneous global state only if certain conditions hold; we called such a vector a partial run. The first lemma here demonstrates that the name "partial run" is justified in that any partial run can indeed be extended to a completed run. Thus, a vector of local states can represent the global system state in the course of an execution if and only if it is a partial run.

Lemma 1. If r is a partial run of system $\mathcal{S}$, then there exists at least one completed run r' of $\mathcal{S}$ that extends it; i.e. for all i, r_i is a prefix of r'_i.

The proof of this first lemma is based on an argument in [Tay89], which is in turn derived from ideas in [LT87]. The proof in [Tay89] assumes that the enabling relations of the processors are finite-branching; i.e. for each processor i and each local state l_i of i, there are only finitely many events $e_i \in E_i$ such that $(l_i, e_i) \in M_i$. This assumption was relaxed in subsequent work [CT90b], and the proof accordingly modified. The proof given below is again a modification of this later proof. The changes arise because we wish to guarantee that every message that is sent is eventually received by the processor for which it is destined; this is not true of the construction given in [CT90b].

Proof: We construct a (possibly infinite) completed run r' from r by iterating through the set of processors and adding appropriately enabled events to the current partial run. We initialize r' to r. For each processor i, we let L_i be a list of all receive events $receive(i,j,m)$ such that $send(j,i,m)$ appears in r'_j, but $receive(i,j,m)$ itself does not yet appear in r'_i. If the system in question is FIFO, we make sure that the receives appear in L_i in the appropriate order, i.e. for each j, the events $receive(i,j,m)$ appear in L_i in the same order as the events $send(j,i,m)$ appear in r'_j. Upon each iteration, for each processor i, we traverse the list L_i until we reach a receive event which is enabled by the current state r'_i; the event is deleted from the list and added to r'_i. We then proceed to the next processor. If there is no such enabled receive event, but there are send or internal events that are enabled by r'_i, then we arbitrarily pick one of these and add it to r'_i. If the added event is $send(i, j, m)$, then we also add the event $receive(j, i, m)$ to the back of the list L_j. Again, we then proceed to the next processor. If there are no enabled events in L_i, and no enabled send or internal events, then r'_i is left unchanged, and we continue directly to the next processor. Iterations through the set of processors continue indefinitely, unless on some iteration no event is added to any local state.

We first show that either the iterations terminate or else every processor is visited infinitely many times during the course of the construction. To do this, we must assure that at each stage, and for each i, the list L_i is finite; otherwise the step "traverse the list L_i" will not terminate. This is certainly the case in the first iteration, since a partial run contains only finitely many events and thus only finitely many sends to any processor. Since L_i consists of receive events for which the corresponding send events are present, L_i is initially finite. At each iteration, at most N send events are added to the partial run, and therefore at most N receive events are added to all the lists L_i together. Thus each L_i remains finite at each stage in our construction.

We must show that this construction results in a completed run. Recall that a partial run is a set of local states such that (1) the ith local state is a feasible state of processor i, (2) the happens-before relation is an irreflexive partial order on the events in the set, and (3) for each receive event there is a corresponding send event in the appropriate local state. Clearly each step of the construction results in a new partial run, since the original r' (r) is a partial run and all added events have the necessary enabling local states and (in the case of receive events) corresponding send events. Also, new events cannot create causal circularity since they do not happen-before any event in the current partial run: the only way an added event could happen-before an event of the current partial run would be for the added event to be $send(i, j, m)$ where $receive(j, i, m)$ is already in r'. This is impossible since there are no unmatched receives in the original r', and no receive is added unless the corresponding send is already present.

It is also necessary to show that the additional constraint on runs is satisfied, i.e. for each i either (1) r'_i is infinite, or (2) there is no event e_i enabled by r'_i, unless it is the reception of a message from j and there is no corresponding send event in r'_j. Suppose r'_i is finite. This implies that after the last iteration of our construction in which r'_i changed, there were no send or internal events enabled by r'_i. Property 2 of enabling relations ensures that therefore all receive events from all neighbours of i are enabled by r'_i. If, during subsequent iterations through the processors, the event $send(j, i, m)$ had been added to r'_j, then the event $receive(i, j, m)$ would have been added to L_i at the same time. Following our construction, $receive(i, j, m)$ would then have been added to r'_i in the following stage. The fact that r'_i did not change implies that L_i remained empty, and so no such send events could appear after the last iteration at which r'_i changed. Therefore the only events enabled by r'_i are indeed receive events for which the corresponding send events are missing. Thus r' is indeed a run.

Finally, we need to show that the run is completed, i.e. that every message that is sent in the run is also received. Note first that if the event $send(j, i, m)$ is added to r'_j at some stage in the construction, then the event $receive(i, j, m)$ will be added to L_i at the same time. We must now ensure that every receive event added to L_i is eventually deleted from it. We have already argued that if r'_i is finite then L_i will be empty, so we need only consider the case where r'_i is infinite. By Property 3 of enabling relations, for each neighbour j of i, there are infinitely many prefixes of r_i at which receives from j are enabled. Since receive events are added to r'_i in preference to send or internal events, this means that infinitely many receive events may be added. Also, the list L_i is always traversed from front to back, and the first enabled receive event that is encountered is added to the local history. Since any element of L_i is preceded in L_i by only finitely many other elements, this guarantees that each receive event will eventually be added.

If the system is FIFO, we must guarantee that all messages are received in the appropriate order. In the initial step, receive events are placed in L_i in the appropriate order; subsequently they are added at the same time as the corresponding send event is added to a local state. Thus for each j, the receive events $receive(i, j, m)$ are added to L_i in the same order as the events $send(j, i, m)$ are added to r'_j. At each step, the first enabled receive encountered in L_i is removed. By Property 1 of enabling relations, it is impossible that the reception of one message from a given processor be enabled, while that of a second message from the same processor is not. Therefore the receive events $receive(i, j, m)$ will be added to r'_i in the correct order for each j. ∎

Note that Properties 2 and 3 of enabling relations are required for this proof. If these properties did not hold of the enabling relations in our system, we would be unable to guarantee that the partial run could be extended to a *completed* run.

If a particular receive event is not at the head of a processor's receive list at one of the finitely many points at which it is enabled, then we will have a message that is sent but not received. In this case, we would have a slightly weaker result: namely, that a partial run of system S_1 could be extended to a completed run of *some* system S_2 with identical processor, channel, and event sets. S_2 need only have enabling relations which are supersets of the corresponding relations in S_1, and which in addition *do* satisfy Properties 2 and 3. In this system the proof would proceed as above.

An immediate consequence of Lemma 1 is the following: if r is a run of system S, $(l_1, ..., l_n)$ is a consistent cut in r, and for some i there is a send event e_i which is enabled by l_i, then there is a run r' of S in which e_i immediately follows l_i. This is true because $(l_1, ..., l_i \cdot e_i, ..., l_N)$ is again a partial run and can therefore be extended to a completed run. Similarly, if e_i is an internal event, or a receive event which does not itself appear in $(l_1, ..., l_N)$ but whose corresponding send does, then there is a run of S in which e_i is the event following l_i.

5.2 Causality and Receive Events

Our second lemma states that the event $receive(j, i, m)$ may occur at processor j as soon as all events of j which happen-before $send(i, j, m)$ have occurred. This is true because of the asynchrony of our systems: message transmission may be arbitrarily fast, and the receiving processor proceed arbitrarily slowly in comparison. The only constraint is that a message cannot actually travel back in time, and be received before some event which precedes its sending.

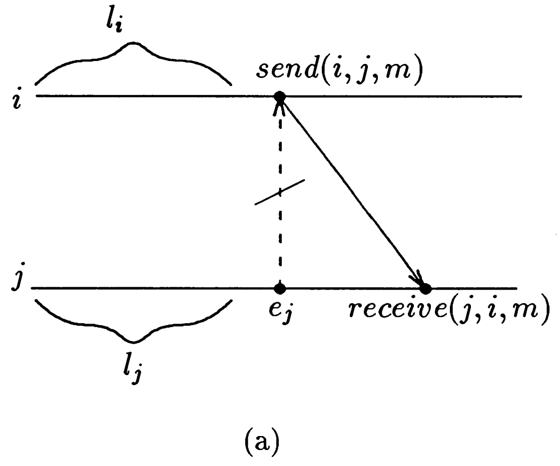


Figure 5.1: Lemma 2. The cancelled dashed arrow indicates "not happens-before" and the solid arrows connect the sending and reception of m.

Lemma 2. Let r be a run such that $l_i \cdot send(i,j,m)$ is a prefix of r_i and $l_j \cdot e_j$ is a prefix of r_j. Suppose that $receive(j,i,m)$ is enabled by l_j and that it has not occurred already in l_j. Then if e_j does not happen-before $send(i,j,m)$, there is a run r' such that $l_i \cdot send(i,j,m)$ is a prefix of r'_i and $l_j \cdot receive(j,i,m)$ is a prefix of r'_j. (See Figure 5.1.)

Proof: The proof proceeds by first showing that there is a partial run r'' of $\mathcal{S}$ where $l_i \cdot send(i,j,m)$ is a prefix of r''_i and $r''_j = l_j$ (without e_j). Then $receive(j,i,m)$ is added to r'', still resulting in a partial run which can be extended to a completed run r' by Lemma 1.

We find a partial run which satisfies the conditions on r'' above, iteratively. (It is not obvious that such a run exists. Perhaps any combination of local states having the required prefixes for i and j will have a message m—potentially involving other processors—which is received but not sent.) Let $MinSends(l_p, q)$ be the minimum local state of processor q in run r which includes the sending of all messages received by p from q in local history l_p. Let $max(l_1, l_2)$, where l_1 and l_2 are local states of a single processor in a single run, be the larger sequence of the two (and therefore encompassing the other). Initialize a vector of local states s as follows: $s_j = l_j$; $s_i = max(l_i \cdot send(i, j, m), MinSends(l_j, i))$; and $s_p = MinSends(l_j, p)$ for p not equal to i or j. We call this the *initial state vector*; of course, this initial set of states is not necessarily a partial run. On each step of the iteration, find a message m' from any processor p to any other processor q such that $receive(q, p, m')$ is in s but $send(p, q, m')$ is not (if such a message does not exist then we are finished). Add events from r_p to s_p until the necessary send is included. In order to meet the conditions of the theorem, we must show that (1) the iteration terminates, and (2) the local state of j is not changed, i.e. upon termination $s_j = l_j$.

We first make an observation to be used extensively in the remainder of the proof: for any inconsistent message m' in the above construction, there is a message chain beginning with m' and ending with a message received within the initial state vector. Consider any message m' as above; either $receive(p, p', m')$ is in the initial state vector, or it is included as the result of adding events until the sending of some m'' is included, so $receive(p, p', m') \rightarrow send(p, p'', m'')$. This argument can be continued resulting in a chain of messages

$$send(p', p, m') \rightarrow receive(p, p', m') \rightarrow send(p', p'', m'') \rightarrow$$

$$receive(p'', p', m'') \rightarrow \ldots \rightarrow receive(p^0, p^n, m^0)$$

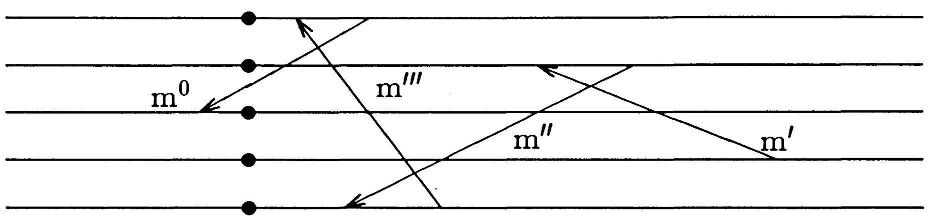


Figure 5.2: Proof of Lemma 2: message chains during iterative construction.

where $receive(p^0, p^n, m^0)$ is in the initial state vector. (See Figure 5.2. The solid circles mark the extent of the initial state vector.)

Suppose that the iteration above never terminates. Then there must be message chains as above where two messages m_1 and m_2 are sent by the same processor. If $send(q, q_1, m_1) \rightarrow send(q, q_2, m_2)$ in the message chain then $send(q, q_2, m_2) \rightarrow send(q, q_1, m_1)$ also because there is a local state which includes $send(q, q', m_2)$ but not $send(q, q', m_1)$ (see Figure 5.3). This cannot occur in any run, hence the iteration terminates. Suppose that during the iteration state s_j is changed, due to a message m': that is, there is a send event $send(j, p, m')$ such that either $e_j = send(j, p, m')$ or $e_j \rightarrow send(j, p, m')$, and $send(j, p, m')$ must be included in s_j. Again, there must be a chain of messages ending with a message m^0 such that $receive(q, q^0, m^0)$ occurs in the original states and $send(j, p, m') \rightarrow receive(q, q^0, m^0)$. There are three cases to consider, depending on what q is. (1) If it is a processor other than i or j, then by the definition of the original set of states s and $MinSends$ there must be a message m_q sent after $receive(q, q^0, m^0)$ from q to j such that $receive(j, q, m_q) \rightarrow e_j$. However, since either $e_j = send(j, p, m')$

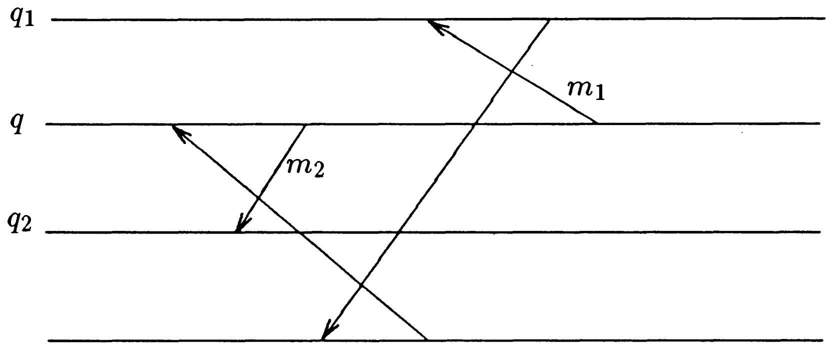


Figure 5.3: Proof of Lemma 2: $send(q, q_1, m_1) \rightarrow send(q, q_2, m_2)$ in the message chain, but $send(q, q_2, m_2) \rightarrow send(q, q_1, m_1)$ at processor q.

or $e_j \rightarrow send(j, p, m')$, we have

$$send(j, p, m') \rightarrow receive(q, q^0, m^0) \rightarrow send(q, j, m_q) \rightarrow$$

$$receive(j, q, m_q) \rightarrow send(j, p, m')$$

and an invalid circularity results. (2) If it is processor j then there is a circularity in the happens-before relation as in the proof of termination above. (3) If it is processor i then either (a) m^0 arrives before $send(i, j, m)$ or (b) m^0 arrives after $send(i, j, m)$ but before the sending of a message m_i to j as in the definition of $MinSends$. In case (a), $send(j, p, m') \rightarrow receive(i, q^0, m^0)$ and $receive(i, q^0, m^0) \rightarrow send(i, j, m)$. However, since either $e_j \rightarrow send(j, p, m')$, or $e_j = send(j, p, m')$, this violates the assumption that $e_j \not\rightarrow send(i, j, m)$. Case (b) is analogous to case (1). Hence the final s_j is equal to l_j.

Let r'' be the final s of the iteration. Then r'' is a partial run of $\mathcal{S}$ such that $r''_j = l_j$ and $l_i \cdot send(i, j, m)$ is a prefix of r''_i. Since $receive(j, i, m)$ is enabled by l_j

in system s, adding event $receive(j,i,m)$ to r''_j still results in a partial run of $\mathcal{S}$. By Lemma 1 this can be extended to a completed run r' as the theorem states. ∎

Lemma 2 does not quite hold for a FIFO system because adding the reception of m could violate the FIFO condition. If, however, there is no message m' that is sent to j prior to m but is not received in l_j, then adding $receive(j,i,m)$ to l_j cannot violate the FIFO condition. This fact, along with the proof of Lemma 2, is sufficient to state the lemma's FIFO counterpart without proof.

Lemma 3. Let run r of FIFO system $\mathcal{S}$ contain events e_j on processor j and $send(i,j,m)$ on processor i, i.e. there are local states l_i and l_j such that $l_i \cdot send(i,j,m)$ is a prefix of r_i and $l_j \cdot e_j$ is a prefix of r_j. Also, let $l_j \cdot e_j$ not contain $receive(j,i,m)$ and let l_j enable $receive(j,i,m)$. If $e_j \not\rightarrow send(i,j,m)$ and there is not a message m' from i to j sent before m but not received in l_j, then there exists a run r' also in $\mathcal{S}$ such that $l_i \cdot send(i,j,m)$ is a prefix of r'_i and $l_j \cdot receive(j,i,m)$ is a prefix of r'_j.

Potential causality between two events at different processors must be transmitted by a message between those two processors, and through the reception of that message. If an event e_i of i happens-before an event e_j of j, then if e_j is not a receive, e_i must happen-before the event immediately preceding e_j as well. This is the conclusion of the fourth lemma.

Lemma 4. Let r be a run, e_i a non-receive event occurring in r_i, and e'_i the event occurring immediately before e_i in r_i. Then for any event e_j occurring in r_j, if $e_j \rightarrow e_i$ it must be the case that $e_j \rightarrow e'_i$.

Proof: Since e_j and e_i are on different processors and e_i is not a receive, e_j cannot happen-immediately-before e_i. Therefore, by the definition of happens-before, there exist events $e_{p_1}, ..., e_{p_n}$ such that $e_j \mapsto e_{p_1} \mapsto ... e_{p_n} \mapsto e_i$. Since e_i is not a receive, only e_i' can happen-immediately-before it. Therefore $e_{p_n} = e_i'$ and $e_j \mapsto e_{p_1} ... \mapsto e_i'$, so $e_j \rightarrow e_i'$. ∎

Repeated applications of Lemma 4 lead to the following result.

Lemma 5. Let r be a run and e_i a non-receive event occurring in r_i. Suppose that $l_i \cdot e_i' \cdot e_i^0 \cdot ... \cdot e_i^n$ is the state immediately preceding e_i in r_i, and that none of the events e_i^0 through e_i^n is a receive event. Then for any event e_j occurring in r_j, if $e_j \rightarrow e_i$ it must be the case that $e_j \rightarrow e_i'$.

Proof: Lemma 4 implies that $e_j \rightarrow e_i^n$, which, together with Lemma 4 again, in turn implies that $e_j \rightarrow e_i^{n-1}$, and so on. ∎

5.3 Local Delay

Our final lemma concerns the following consequence of local delay: if a set of events has been locally delayed in the states of a partial run, then in some extending run, each event is re-enabled with no intervening receives. Note that this is not immediate from the definition of local delay: that definition guarantees only that for each delayed event, there is some run where no communication is required to end the disabling. We need to show that it is not the case that in each of these runs, some other event must wait upon reception of a message to be re-enabled.

Lemma 6. Let r be a protocol run; $(l_1, \ldots, l_N)$ be a consistent cut of r; $p_1, \ldots, p_k$ be (not necessarily distinct) processors; and events $e_{p_1}, \ldots, e_{p_k}$ be locally delayed by the protocol in states $l_{p_1}, \ldots, l_{p_k}$, respectively. Then there is a completed run r' such that $(l_1, \ldots, l_N)$ is a consistent cut of r', and, for $i = 1, \ldots, k$, there are no receive events between l_{p_i} and a state in which e_{p_i} is no longer disabled.

Proof: First we apply the definition of local inhibition. There must exist, for each p_i, a run r^{p_i} containing an extension l'_{p_i} of l_{p_i}, such that $l'_{p_i} - l_{p_i}$ contains no receive events and e_{p_i} is not disabled in l'_{p_i}. Let $e_{i_1}, e_{i_2}, \ldots, e_{i_n}$ denote the subsequence of non-receive events between l_{p_i} and l'_{p_i}. In order for run r^{p_i} to exist, there must be pairs (l_{p_i}, e_{i_1}), $(l_{p_i} \cdot e_{i_1}, e_{i_2})$, and so on in the enabling relation for i. Consequently, we can add each of these events to the consistent cut $(l_1, \ldots, l_N)$, resulting in a partial run of the system. (Note that, because these events are not receives, there is no causal circularity introduced, and no receive event is added for which the corresponding send will be missing.) This generalizes to all other processors. Since any partial run of a system can be extended to a completed run of the system, by Lemma 1, the resulting completed run satisfies the conditions for r' in the theorem statement. ∎

5.4 Summary

In this chapter, we have presented and proved six lemmas. The first demonstrates the correspondence between partial and completed runs, and is used in the proofs of three of the remaining five lemmas. These three lemmas examine the relationships between the various runs of a given system. The remaining two lemmas trace the connection between receive events and potential causality. All these lemmas will be used extensively in the proofs of our impossibility and complexity results in the following chapters.

Chapter 6

Impossibility Results for Non-FIFO Systems

In this chapter we present impossibility results for the non-FIFO cases in Figure 4.2 of Chapter 4. According to the claims made in this figure, global send inhibition is necessary (and sufficient) for a consistent-cut protocol for non-FIFO systems, regardless of whether or not protocol messages may be inconsistent in the designated cut. To prove our claim, we assume the possible use of the maximum degree of inhibition other than global send inhibition: that is, global receive inhibition and local send inhibition. We then show that these are not sufficient for a protocol to produce cuts in which system messages are guaranteed to be consistent. We do this by displaying a possible run of the protocol in which there is an inconsistent system message. The impossibility results for the remaining non-FIFO cases then follow immediately, as in these cases the degree of inhibition involved is weaker and/or the requirements on the designated cut are stronger.

Given that there is no consistent-cut protocol with a particular kind and degree of inhibition which is successful for all non-FIFO systems, we might wish to moderate our demands, and consider whether there are CCPs with the requisite characteristics that are successful for some subset of non-FIFO systems. For instance, might there be a consistent-cut protocol using, say, local send inhibition only, which is correct for all non-FIFO systems in which the processor network has a tree structure? If we specify a class of systems by some physical property of the processor network, then we can show that, once again, there is no consistent-cut protocol using up to global receive inhibition and local send inhibition that is successful for any class of non-FIFO systems.

6.1 The Impossibility Theorem

The theorem presented below corresponds to the third IMP† case in Figure 4.2; all other impossibility results for non-FIFO systems in Figure 4.2 are then direct consequences of this theorem.

Theorem 1. Using up to global receive inhibition and local send inhibition, there is no consistent-cut protocol for non-FIFO systems. Moreover, this is true even if we require only that system messages be consistent.

In order to prove Theorem 1, we need only prove that, for any protocol satisfying the conditions of the theorem, in some run of the protocol with respect to some system, the cut produced by the protocol is inconsistent. We can therefore make worst-case assumptions about the enabling relations on sequences of system events. We then show that, for any CCP as described, there is necessarily a run with the following property: for some two processors, there is causal circularity

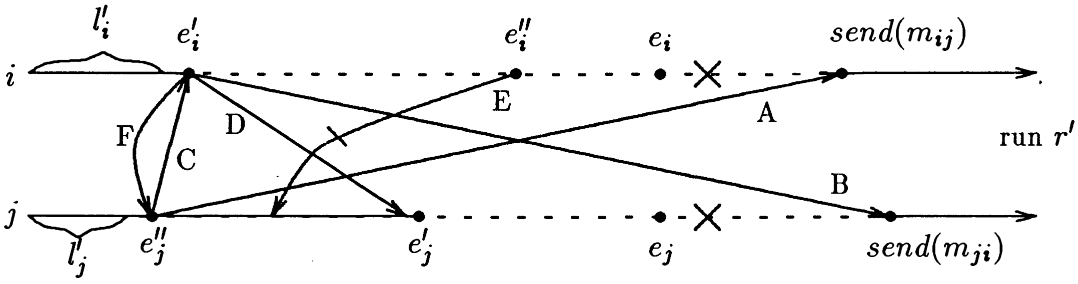


Figure 6.1: Proof of Theorem 1. Dashed lines are intervals in which no receive events occur, arrows represent happens-before, and 'X' marks cut states. (A),(B) follow from Lemma 2. (C),(D) then follow from Lemma 4. (D) and absence of (E) imply (F).

between the last states preceding their respective cuts in which they are willing to receive messages from each other.

Proof of Theorem 1: Given a run r of the protocol, let $l_p \cdot e_p$ be the cut state of p in r for each processor p. Let i be any processor. Let l_i' be the last proper prefix of i's cut state in which receives from some processor are enabled and e_i' be the event following l_i' in r. Let j be any processor receives from which are enabled in l_i'; from the manner in which l_i' was chosen, there must be at least one such processor. Let l_j' be the last proper prefix of j's cut state in which receives from some processor are enabled and e_j' be the event following l_j' in r. Let l_j'' be the last proper prefix of $l_j' \cdot e_j'$ in which receives from i are enabled and e_j'' be the event following l_j''. (See Figure 6.1. Note that some of the receive-free intervals may be empty; e.g., it may be that $e_i = e_i'$.)

Let $send(i,j,m_{ij})$ be a send event of processor i that is enabled by the subsequence of system events in $l_i \cdot e_i$. Let $send(j,i,m_{ji})$ be a send event of processor j that is enabled by the subsequence of system events in $l_j \cdot e_j$. The protocol may inhibit the occurrence of $send(i,j,m_{ij})$ in $l_i \cdot e_i$ and of $send(j,i,m_{ji})$ in $l_j \cdot e_j$. However, since this inhibition can only be local, by Lemma 6 there is a run r' in which $l_p \cdot e_p$ is a prefix of r'_p for each p, and no receive events occur between $l_i \cdot e_i$ (respectively $l_j \cdot e_j$) and the state in which $send(i,j,m_{ij})$ (respectively $send(j,i,m_{ji})$) is re-enabled. We can also assume that $send(i,j,m_{ij})$ and $send(j,i,m_{ji})$ occur as soon as they are re-enabled. We consider r'.

We know that the event $receive(i,j,m_{ji})$ does not occur in l'_i; if it did, the cut would be inconsistent. We also know that $receive(i,j,m_{ji})$ is enabled by l'_i, by the definition of l'_i. Suppose that $e'_i \not\rightarrow send(j,i,m_{ji})$. Lemma 2 then implies that there is some run r'' in which $l'_i \cdot receive(i,j,m_{ji})$ is a prefix of r''_i and $l_j \cdot e_j \cdot \ldots \cdot send(j,i,m_{ji})$ is a prefix of r''_j. In this run, neither l'_i nor any prefix of l'_i can be i's cut state: if it were, the same local state would be i's cut state in run r' (by condition (2) of the definition of CCP), which is not the case. Also by condition (2), $l_j \cdot e_j$ must be j's cut state in r''. In sum, r'' is a possible run in which the message m_{ji} is sent after j's cut state and received before i's, and the cut produced by the protocol is inconsistent. Thus the assumption that $e'_i \not\rightarrow send(j,i,m_{ji})$ in run r' leads to a contradiction. Therefore it must be the case that $e'_i \rightarrow send(j,i,m_{ji})$. A symmetric argument establishes that $e''_j \rightarrow send(i,j,m_{ij})$. (See arrows (A) and (B) in Figure 6.1).

Recall that no receive events occur in j's cut state after e'_j, since l'_j is the last prefix of j's cut state in which receives from any processor are enabled.

Also, by the choice of r'', no receive events occur between e_j and $send(j,i,m_{ji})$. By applying Lemma 5, we have that $e'_i \rightarrow send(j,i,m_{ji})$ implies $e'_i \rightarrow e'_j$. Similarly, there are no receive events after e'_i and before $send(i,j,m_{ij})$. Therefore if $e''_j \rightarrow send(i,j,m_{ij})$, it must be the case that $e''_j \rightarrow e'_i$. (See arrows (C) and (D) in Figure 6.1.)

We claim that there can be no message sequence starting between e'_i and e_i and ending between e''_j and e'_j (the cancelled arrow (E) in Figure 6.1). If we can show this, then $e'_i \rightarrow e'_j$ implies $e'_i \rightarrow e''_j$ (arrow (F)) and the happens-before partial order will be violated. So we suppose that there is such a message sequence. It must have length at least two, since receives from i are disabled at j in the interval of interest. Suppose the first event of the message sequence is $e''_i = send(i,k,m)$ to some neighbour k of i. The event $receive(k,i,m)$ must occur in k's cut state; if it did not, then the remainder of the message sequence will start after k's cut state and end before j's cut state, thereby necessarily including an inconsistent message. We can assume that either the event immediately after $send(i,j,m_{ij})$ is a system send event $send(i,k,m_{ik})$ to k, or that system sends to k have been disabled by the protocol in the state $l_i \cdot e_i \cdot \ldots \cdot send(i,j,m_{ij})$. In the latter case, we can consider (again by Lemma 6, since the delay can only be local) a run which is identical to r' up through $send(i,j,m_{ij})$, $send(j,i,m_{ji})$ and k's cut state, and in which there are no receive events occurring between $send(i,j,m_{ij})$ and the state in which some system send to k is re-enabled. Assume that the event following this state is $send(i,k,m_{ik})$. In either case—$send(i,k,m_{ik})$ immediately following $send(i,j,m_{ij})$, or after a sequence of non-receive events—necessarily $receive(k,i,m) \not\rightarrow send(i,k,m_{ik})$. Otherwise, since there are no receive events between $e_i'' = send(i, k, m)$ and $send(i, k, m_{ik})$, $receive(k, i, m) \rightarrow send(i, k, m_{ik})$ would imply (by Lemma 5) that $receive(k, i, m) \rightarrow e_i'' = send(i, k, m)$, resulting in circular causality. Now, since $receive(k, i, m) \not\rightarrow send(i, k, m_{ik})$, we can apply Lemma 2 and produce a run in which the message m_{ik} is inconsistent. Therefore, by contradiction, there is no message sequence beginning between e_i' and e_i and ending between e_j'' and e_j'.

Thus $e_i' \rightarrow e_j'$ implies $e_i' \rightarrow e_j''$. We have already shown that $e_i' \rightarrow e_j'$ and that $e_j'' \rightarrow e_i'$; consequently, there is causal circularity in run r'. This of course is impossible. Therefore it must be the case either that $e_i' \not\rightarrow e_j'$ and hence $e_i' \not\rightarrow send(j, i, m_{ji})$ or that $e_j'' \not\rightarrow e_i'$ and hence $e_j'' \not\rightarrow send(i, j, m_{ij})$. Suppose for definiteness that it is the former. Again we invoke Lemma 2 and condition (2) of Definition 22 to show that there is a run in which the message m_{ji} is inconsistent. Since m_{ji} is a system message, our proof is now complete. ∎

We noted in Chapter 4 that if one allows the selective reception of messages from a given neighbour of a processor, then there is a CCP using global receive inhibition which is successful for non-FIFO systems. The proof of our impossibility theorem should therefore require the non-selectivity assumption; and in fact it does. For instance, the argument that if $receive(k, i, m)$ can occur in k's cut state then $receive(k, i, m_{ik})$ can occur at the same point, is no longer valid.

We have shown that a consistent-cut protocol must use global send inhibition if it is to be successful for non-FIFO systems in general. One might next ask whether there is some class of non-FIFO systems for which CCPs with lesser degrees, or different kinds, of inhibition might exist. The answer, however, is negative. If we specify a class of systems by some property of the processor network, such as the number of processors or the network topology, then there is no class of non-FIFO systems for which anything less than global send inhibition suffices.

Corollary 1. Let $\mathcal{K}$ be a class of non-FIFO systems $S = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$, where membership in $\mathcal{K}$ is based solely on properties of the sets $\mathcal{I}$ and $\mathcal{C}$. Then there is no consistent-cut protocol using up to global receive inhibition and local send inhibition that is successful for all systems S in $\mathcal{K}$. Moreover, this is true even if we require only that system messages be consistent.

Proof: The only network characteristic that was exploited in the proof of Theorem 1 is that there be a pair of neighbouring processors. This is true of every system in $\mathcal{K}$, as a system is required by definition to have a connected channel set. Since $\mathcal{K}$ contains all systems having the required network property, we can again make worse-case assumptions about the enabling relations on sequences of system events. The proof then proceeds exactly as for Theorem 1. ∎

To be strictly accurate, we should rule out the possibility that $\mathcal{K}$ consists of all non-FIFO systems in which there is only one processor. Since the claim of such a system to be "distributed" is disputable, we ignore this caveat and state our corollary as above.

6.2 Summary

We have shown that global send inhibition is necessary in order for a consistent-cut protocol to be successful for non-FIFO systems: using up to global receive inhibition and local send inhibition, there is no such protocol. This result holds whether or not protocol messages would be allowed to be inconsistent. The argument used to prove Theorem 1 suffices to prove an even stronger result: that global send inhibition is required when designing a consistent-cut protocol for any class of non-FIFO systems. Theorem 1 also suffices to imply the remaining impossibility results of Figure 4.2. The impossibility results for FIFO systems will be presented in the following chapter.

Chapter 7

Impossibility Results for FIFO Systems

This chapter contains the theorems corresponding to the two IMP† results for FIFO systems in Figure 4.2 of Chapter 4:

Theorem 2. There is no non-inhibitory CCP for FIFO systems, even if protocol messages are not required to be consistent.

Theorem 3. There is no CCP for FIFO systems that local send inhibition and no receive inhibition, if protocol messages are required to be consistent.

Recall from Chapter 4 that there is a consistent-cut protocol ($Flood1$) for FIFO systems which uses local send inhibition; however, $Flood1$ does produce cuts in which there are inconsistent protocol messages. Theorem 3 asserts that it is impossible to prevent this phenomenon using local send inhibition only. Theorem 2 demonstrates that some form of inhibition is always required, whether or not one is concerned with the consistency of protocol messages. The remainder of the impossibility results for FIFO systems as illustrated in Figure 4.2 follow immediately from Theorems 2 and 3.

The proofs of these theorems are fairly similar: that of Theorem 2 first appeared in [Cri89], and that of Theorem 3 in [CT90b]. We present the proof of Theorem 3 first here, as it is slightly the more involved of the two. We then discuss the modifications of this argument required to prove Theorem 2. In both proofs, we assume the existence of a CCP with the required characteristics and derive a contradiction. The proofs differ in that in the non-inhibitory case, one can assume that the event immediately following a cut state is a system send, while in the presence of local send inhibition, one cannot. The proof of Theorem 2 produces a possible run in which one of these system messages is inconsistent. The proof of Theorem 3 also produces a possible run in which there is an inconsistent message; however, due to local send inhibition, this message can be made to be a protocol message. This is the intuition behind $Flood1$.

The arguments of this chapter are more complicated than those of the impossibility proof for non-FIFO systems. The extra difficulty arises because one cannot as easily "push back" the reception of messages in a run. In the non-FIFO case, it is sufficient to show that a causal relationship could not hold between particular events; here, we must ensure in addition that the FIFO condition is not violated. Thus the fact that the system is FIFO can be made to work for a hypothetical protocol, and against our attempt to derive a contradiction.

A sketch of the development of our proof of Theorem 3 is as follows. We assume that there is a CCP for FIFO systems which utilizes local send inhibition only, and which guarantees that both system and protocol messages are consistent. Through a sequence of lemmas we derive a contradiction. First we show that if event e_i is the last event of i's cut state and event e_j is in j's cut state, then $e_i \rightarrow e_j$ implies that in fact e_i and e_j are the sending and receiving of a single message, and e_j is the last event of j's cut state (Lemma 8). Next we show that, if such a message does not occur from i to j, then there must be an "essential" message that j sends to i before reaching its cut state; the reception of this message causes i's cut state to occur if it has not previously done so (Lemma 9). Next, by Lemmas 10 through 13, we prove the existence of a run in which no cut states occur along a single message as in Lemma 8 and therefore every pair of processors must exchange essential messages; furthermore, for one processor, say k, all of the essential messages which it sends arrive after its neighbours' cut states. Finally, our argument concludes with a proof that there is a possible run as in Lemma 13, but in which k is forced to reach its cut state before sending an essential message to one processor. The absence of this message allows an inconsistent cut to be produced by the protocol.

7.1 Preliminary Lemmas

We now proceed to the sequence of lemmas required for the proof of Theorem 3. Throughout the remainder of this chapter, we assume that systems are FIFO. Furthermore, we assume that there exists a consistent-cut protocol for FIFO systems that uses only local send inhibition, and that guarantees the consistency of all messages. Again, we can make worst-case assumptions about the enabling relations on sequences of system events. In particular, for simplicity we assume that all system receives are always enabled.

The following lemma states that if one processor reaches its cut state immediately upon (after) sending a message, then if the receiving processor has not already reached its cut state, it must do so upon reception of that message. If it does not, then it leaves open the possibility of a post-cut message being received before the cut state is reached.

Lemma 7. Let r be a run of the protocol, and suppose that j's cut state in r occurs upon the send of a message to i. Then i's cut state in r must occur before the corresponding receive, or immediately after.

Proof: Suppose j's cut state is $l_j \cdot \mathit{send}(j,i,m)$, and i's cut state does not occur upon or before $\mathit{receive}(i,j,m)$, i.e. there is at least one event after $\mathit{receive}(i,j,m)$ which is contained in i's cut state. Let l_i be the prefix of i's local history up to and including $\mathit{receive}(i,j,m)$, and e_i be the event immediately following $\mathit{receive}(i,j,m)$. We assume that some system send to i is enabled by the subsequence of system events in j's cut state. Suppose this send event is disabled by the protocol for some (possibly zero) number of states following j's cut state. By Lemma 6, we know there is some run r' with identical cut states and in which there are no receive events between j's cut state and the re-enabling of this send event. We can also assume that this send occurs as soon as it is re-enabled. Let $\mathit{send}(j,i,m')$ be the first send to i occurring after j's cut state (i.e. m' is either a protocol message sent in the receive-free interval following j's cut state, or it is the system message which we have assumed occurs at the end of this interval—see Figure 7.1 (a).)

If $e_i \rightarrow \mathit{send}(j,i,m')$ then by Lemma 5 and the fact that no receives occur between $\mathit{send}(j,i,m)$ and $\mathit{send}(j,i,m')$, we have $e_i \rightarrow \mathit{send}(j,i,m)$. However,

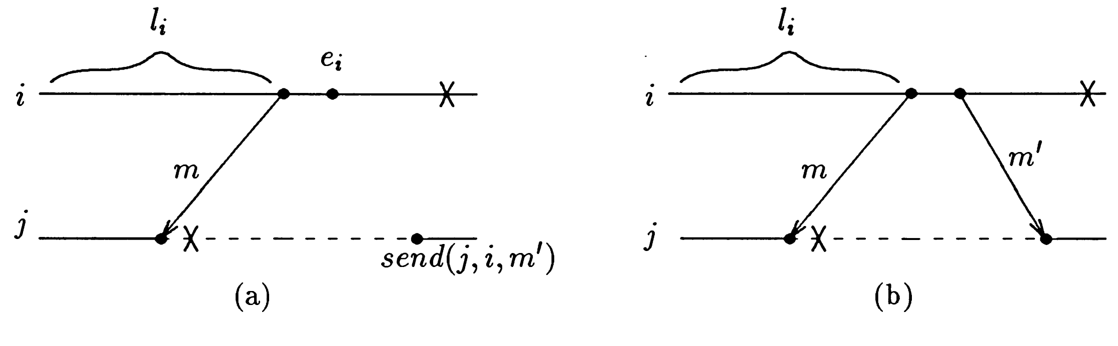


Figure 7.1: Proof of Lemma 7: (a) run r'; (b) run r''. Dashed lines represent receive-free intervals.

$send(j,i,m) \rightarrow receive(i,j,m) \rightarrow e_i$, resulting in causal circularity; therefore, it must be that $e_i \not\rightarrow send(j,i,m')$. Also, since the system is FIFO and m is received at i before e_i occurs, all messages sent from j to i before m' have been received before the occurrence of e_i. Therefore by Lemma 3, there is a run r'' in which $l_i \cdot receive(i,j,m')$ is a prefix of r''_i, and $l_j \cdot send(j,i,m) \cdot \ldots \cdot send(j,i,m')$ is a prefix of r''_j (see Figure 7.1 (b)). The cut state of i in run r'' cannot occur before $receive(i,j,m')$, or it would have done so in run r' (condition (2) of the definition of CCP). Therefore in run r'', m' is sent after j's cut state and received before i's, and the cut is inconsistent, which is a contradiction. ∎

The next lemma states the circumstances under which the last event of one cut state can happen-before some event in another processor's cut state: namely, the two events must be the send and receive of a single message, and the latter event must be the last event in the cut state of the processor at which it occurs.

Lemma 8. Let r be a run of the protocol. Suppose that i's cut state in r is

$l_i \cdot e_i$, and $e_i \rightarrow e_j$ where e_j is an event in j's cut state. Then e_i is $send(i,j,m)$ and e_j is $receive(j,i,m)$ for some message m, and e_j is the last event of j's cut state.

Proof: Since $e_i \rightarrow e_j$, there must be a sequence of messages $m_0,\ldots,m_n$ such that

1. $e_i \rightarrow send(i,p_0,m_0)$ or $e_i = send(i,p_0,m_0)$;

2. $receive(p_t,p_{t-1},m_t) \rightarrow send(p_t,p_{t+1},m_{t+1})$ for each t; and

3. $receive(j,p_{n-1},m_n) \rightarrow e_j$ or $e_j = receive(j,p_{n-1},m_n)$.

Suppose that $e_i \neq send(i,p_0,m_0)$. Then $send(i,p_0,m_0)$ occurs after i's cut state; however, e_j and therefore $receive(j,p_{n-1},m_n)$ occur before j's cut state. This implies that one of the messages $m_0,\ldots,m_n$ is inconsistent. Therefore $e_i = send(i,p_0,m_0)$. By Lemma 7, processor p_0 must reach its cut state before or immediately upon reception of m_0. Consequently, we must have $m_0 \equiv m_n$: otherwise, $m_1,\ldots,m_n$ would contain an inconsistent message. Now, we know that either $receive(j,i,m_0) \rightarrow e_j$ or $e_j = receive(j,i,m_0)$ (see (3) above). Again by Lemma 7, j's cut state must occur, at the latest, immediately after $receive(j,i,m_0)$. Thus $receive(j,i,m_0) \rightarrow e_j$ would imply that e_j is not an event of j's cut state, contradicting the assumptions of the lemma. Therefore, $e_j = receive(j,i,m_0)$, and is the last event of j's cut state. ∎

7.2 Primary Lemmas

The question of whether a pair of processors' cut states occur as in Lemma 8, i.e. immediately upon the sending and reception of a single message, is a critical one. If they do, then there is no possibility of an inconsistent message between the processors: any such message would violate either the FIFO condition or the happens-before partial order. If they do not, then there could potentially be an inconsistent message between the processors. We therefore use the following terminology: in a run r, i's and j's cut states *occur along a message* if there is some message m such that i reaches its cut upon $send(i, j, m)$ and j reaches its cut upon $receive(j, i, m)$, or vice versa. A processor j's cut state *occurs along a message* if there is some i such that i's and j's cut states occur along a message.

Lemma 9 states conditions which must be true of the cut states of processors which do not cut along a message.

Lemma 9. Let r be a run of the protocol in which i's and j's cut states do not occur along a j to i message. Let $l_i \cdot e_i$ be i's cut state in r. Then $l_i \cdot e_i$ must contain an event $send(i, j, m)$ such that, in all runs r' of the protocol in which $l_i \cdot e_i$ is i's cut state—and in which i's and j's cut states do not occur along a j to i message—either $receive(j, i, m)$ is the last event in j's cut state, or $receive(j, i, m)$ occurs after j's cut state.

Proof: Let j's cut state in run r be $l_j \cdot e_j$ and suppose that there are no sends to j included in i's cut state in r. If some send event from i to j has been disabled by the protocol in i's cut state, then by Lemma 6 there is a run r' in which cut states remain the same and the re-enabling of this send event occurs without intervening receives. We can assume that the send occurs immediately upon its re-enabling. Let m' be the first message sent from i to j after i's cut state. (As in Lemma 7, m' is either a protocol message sent during the receive-free interval, or the system message that we have assumed is sent at the end of this interval.)

There are no receives between e_i and the sending of m'. Furthermore, it is the case that $e_j \not\rightarrow send(i,j,m')$, because by the preceding observation and Lemma 5, $e_j \rightarrow send(i,j,m')$ would imply $e_j \rightarrow e_i$; however, by Lemma 8 this could only happen if $e_j = send(j,i,m'')$ and $e_i = receive(i,j,m'')$, which we have assumed not to be the case. So $e_j \not\rightarrow send(i,j,m')$. Since there are no messages sent from i to j in $l_i \cdot e_i$, the hypotheses of Lemma 3 are satisfied, and therefore there exists a run r'' in which $receive(j,i,m')$ occurs immediately after l_j. In run r'', j's cut state cannot occur before $receive(j,i,m')$; therefore the cut produced in run r'' is inconsistent.

So i must send at least one message to j in $l_i \cdot e_i$. Let m be the last such message. Assume there is a run in which i's cut state is $l_i \cdot e_i$, i's and j's cut states do not occur along a j to i message, and $receive(j,i,m)$ occurs before e'_j, the last event of j's cut state. As above, if system sends from i to j have been disabled by the protocol in i's cut state, we can consider a possible run r' and message m' such that: r' has the same cut states as r, m' is the first message sent by i after its cut state, and there are no receive events between i's cut state and $send(i,j,m')$. Since i's and j's cut states do not occur along a j to i message, $e'_j \not\rightarrow e_i$ so $e'_j \not\rightarrow send(i,j,m')$ (by Lemma 5, since there are no intervening receives). All messages sent to j in $l_i \cdot e_i$ have been received before e'_j (since m' is the last such message, and $receive(j,i,m')$ precedes e'_j), and so applying Lemma 3 produces a run in which the cut is inconsistent. Consequently, either $e'_j = receive(j,i,m)$ or $receive(j,i,m)$ occurs after j's cut state. ∎

We call a message such as that described in Lemma 9 *essential* (i.e. an essential message is one upon whose reception the receiving processor will reach its cut state, if it has not already done so). If a message is not essential, but in the run under consideration it happens to be received after or immediately before the receiver's cut, the message is termed *incidental.* We have seen that every pair of neighbours must either have cut states occurring along a message or exchange essential messages in order to guarantee consistency of the cut. When essential messages are exchanged, Lemma 10 demonstrates that the reception of certain messages can be "pushed back" until the receiving processor necessarily reaches its cut state immediately after the reception of one of them. Later, in order to complete the proof of our theorem, we demonstrate a run in which this "pushing back" results in a processor reaching its cut state before it has sent all the necessary essential messages.

Lemma 10. Let r be a run of the protocol in which there are processors i and j whose cut states do not occur along a j to i message. Let $l_p \cdot e_p$ be the cut state of processor p in run r for all p. Then there exists a sequence of messages $m_1, \ldots, m_k$ sent to j in $l_i \cdot e_i$ such that in some run r', $l_j \cdot receive(j, i, m_1) \cdot \ldots \cdot receive(j, i, m_k)$ is j's cut state. Moreover, it is possible to find an r' for which, in addition, the following two properties hold:

1. for all p such that $e_j \not\rightarrow e_p$ (i.e. j's and p's cut states do not occur along a j to p message in r—note that processor i falls into this case), $l_p \cdot e_p$ is p's cut state in r';
2. for all p such that $e_j \rightarrow e_p$ (i.e. j's and p's cut states do occur along a j to p message in r), l_p is a prefix of p's cut state in r'.

Proof: Let m_1 be the first message sent to j in $l_i \cdot e_i$ but not received in l_j. Let $m_2, \ldots, m_n$ be any other such messages. By Lemma 9, $m_1, \ldots, m_n$ must contain an essential message. If $e_j = receive(j, i, m_1)$, then r itself satisfies the lemma. Otherwise, let s^1 be the vector of local states where (a) $s^1_j = l_j \cdot receive(j, i, m_1)$, (b) $s^1_p = l_p \cdot e_p$ for p as in (1) in the statement of the lemma, and (c) $s^1_p = l_p$ for p as in (2). Then s^1 is a partial run as follows. Firstly, each event of each processor is enabled by the preceding local state, since $(l_1 \cdot e_1, \ldots, l_N \cdot e_N)$ is a partial run, and the only additional event is $receive(j, i, m_1)$. Secondly, $receive(j, i, m_1)$ is the only receive event added, and $send(i, j, m_1)$ occurs in $l_i \cdot e_i$; also, the only send event deleted from the events in $(l_1 \cdot e_1, \ldots, l_N \cdot e_N)$ is (possibly) e_j, and any corresponding receives have been removed as well (see Lemma 8 and (2) in the statement of the current lemma). Thus there is no receive event in s^1 for which the corresponding send is missing. Finally, the happens-before partial order is not violated: $receive(j, i, m_1)$ does not happen-before anything in s^1, so no circularity is introduced. Thus, s^1 is a valid partial run. The FIFO condition is not violated, as m_1 is the first i to j message not received in l_j.

The partial run s^1 can be extended by Lemma 1 to a total run r^1. (See Figure 7.2 (b). Event e_p in (a) follows l_p in (b) if $e_j \not\rightarrow e_p$.) If j's cut state in r^1 is $l_j \cdot receive(j, i, m_1)$, then r^1 satisfies the lemma. Otherwise, let s^2 be the vector of local states such that (a) $s^2_j = l_j \cdot receive(j, i, m_1) \cdot receive(j, i, m_2)$, (b) $s^2_p = l_p \cdot e_p$ for p as in (1), and (c) $s^2_p = l_p$ for p as in (2). By a similar argument to that above (a slightly simpler one, as s^2 is an extension of s^1), s^2 is also a partial run. Since $m_1, \ldots, m_n$ contain an essential message m_k, we can repeat this process to obtain a run satisfying the lemma, in k or fewer steps (Figure 7.2 (c)). ∎

The following three lemmas show that there exists a possible run of the protocol in which one processor sends an essential message to each of its neighbours,

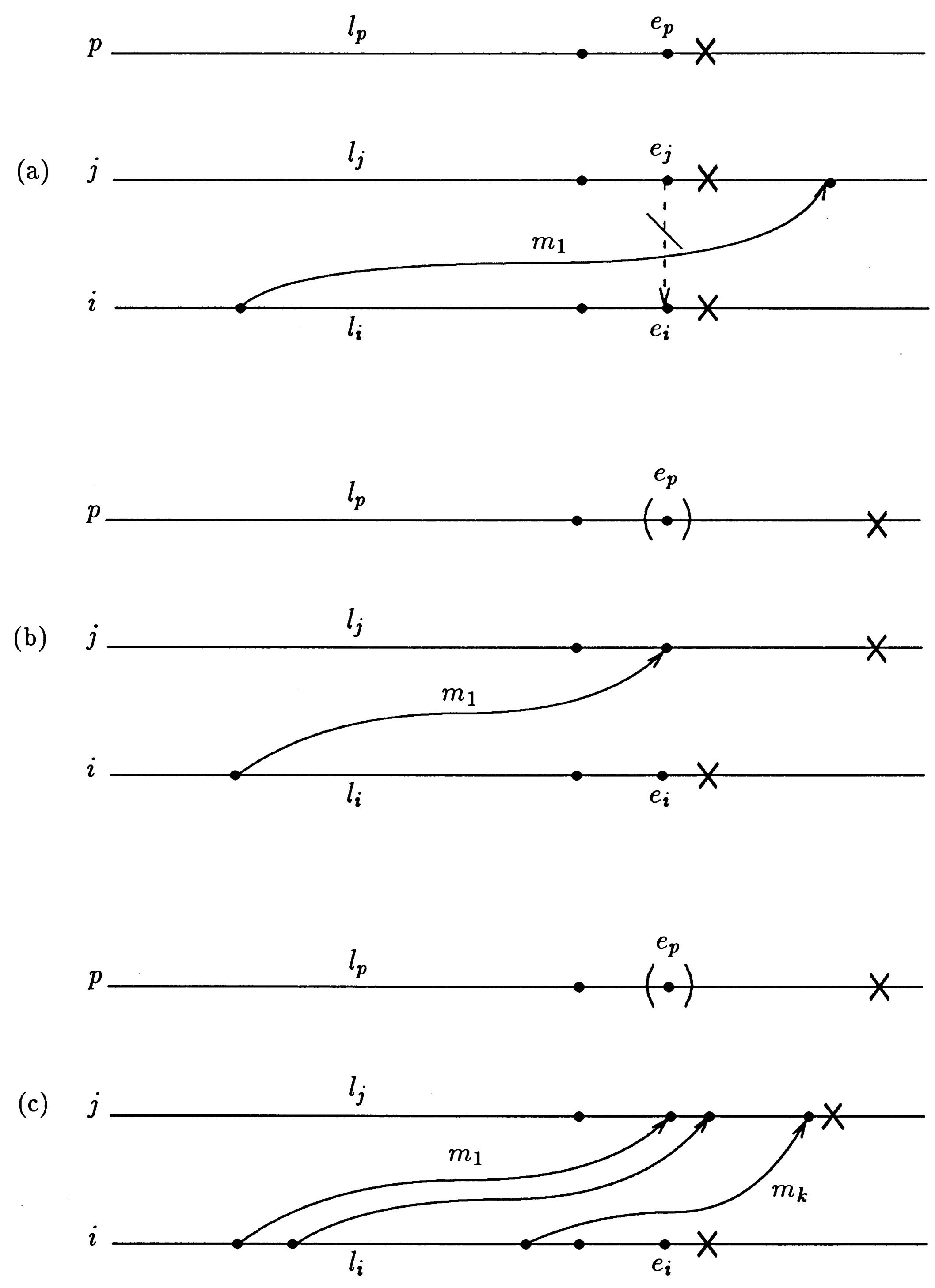


Figure 7.2: Proof of Lemma 10: (a) the original run r; (b) after the first step, run r^1; (c) the final run r'.

but all neighbours reach their cut state before that message is received. We start with an arbitrary run, and progressively massage it into a run satisfying these criteria. The final argument in the proof of Theorem 3 will then show how this run can be manipulated to form a possible run in which the cut is inconsistent.

We start by showing that it is possible to find a run in which at least one processor's cut state ends with a receive event. Below, the *degree* of a processor in a network refers to the number of its neighbours. Recall that a protocol must apply to systems with any network configuration (processor set and channel set).

Lemma 11. There exists a protocol run r in which there is a processor j such that $degree(j) \geq 2$ and j's cut state in r occurs upon a receive event.

Proof: Let S be any system such that the processor network contains two neighbouring processors i and j each with degree at least two. Suppose there exists a run r of the protocol with respect to S in which neither i's nor j's cut state occurs upon a receive; therefore, they do not occur along a j to i message and i sends an essential message to j. Lemma 10 can then be applied, and produces a run in which j's cut state occurs upon receipt of a message from i. ∎

Next, we show that it is possible to find a run in which one processor's cut state occurs upon a receive but not along a message.

Lemma 12. There exists a protocol run in which there is a processor whose cut state occurs upon reception of a message but not along a message (i.e. a processor i such that the last event of i's cut state is $receive(i, j, m)$ but $send(j, i, m)$ is not the last event of j's cut state).

Proof: By Lemma 11 there is a run r and a processor j of degree at least two such that j's last cut-state event in r is a receive. Let $l_j \cdot e_j$ be j's cut state. Let i be a neighbour of j such that e_j is not the receive of a message from i; it is possible to find such an i since $degree(j) \geq 2$. By Lemma 9 there is an essential message sent to i in $l_j \cdot e_j$; since e_j is not a send, this message must be sent in l_j. Let m be the first j to i message that is incidental or essential in run r. If i's last cut-state event e_i is $receive(i, j, m)$ then i satisfies the lemma. If not, one can apply Lemma 10 to get a run in which i's cut state does occur upon reception of a message from j and in which $l_j \cdot e_j$ is j's cut state. In this new run, i satisfies the lemma. ∎

The third step is to find a run in which each processor's last cut-state event is the reception of a message not along the cut, and there is one processor which has sent none of these messages.

Lemma 13. There exists a protocol run in which

1. every processor reaches its cut state upon a receive but not along a message; and
2. there is a processor j such that none of j's neighbours reaches its cut state upon receiving a message from j.

Proof: By Lemma 12, there is a protocol run r in which at least one processor i satisfies the condition described in (a). If i's last cut-state event is $receive(i, p, m)$, set $D = \{i\}$, and let Q be a queue consisting of all neighbours of i, with p at the head. Since i's and p's cut states do not occur along a p to i message, there exists (by Lemma 10) a run r^1 in which p's cut state occurs upon reception of a message from i, and in which i's cut state remains unchanged. Add p to D, delete it from Q, and add its neighbours other than i to the back of Q. Repeat this procedure: at any step, the processors in D are amongst those which, in the current run r^{t-1}, satisfy condition (1). In addition, the processor p' at the head of Q is a neighbour of a processor $k \in D$, and p''s and k's cut states do not occur along a p' to k message. Lemma 10 again gives a run r^t of S in which p''s cut state occurs upon reception of a message from k, and for all $d \in D$, d's cut state in r^t is identical to that in r^{t-1}. Processor p' satisfies the condition of (1) in run r^t; add it to D, delete it from Q, and add all its neighbours which are neither in D nor already in Q to Q. Since at each step one processor is added to D, this procedure must eventually halt. Since the network is connected, all processors will eventually be added to Q and thus to D. Therefore the repetition ends with a run r^t satisfying (1).

Let j be the last processor removed from Q; all of j's neighbours must be in D. However, after the first iteration of the above construction, any processor in D reaches its cut state upon reception of a message from some other processor in D. Thus j satisfies (2). ∎

7.3 Proof of Theorem 3

By the final lemma above, we can find a run r and a processor j such that j's cut state in r occurs upon a receive but not along a message, and such that all of j's neighbours reach their cuts upon reception of messages sent by processors other than j. Let p's cut state in run r be $l_p \cdot e_p$ for each processor p. Since j's cut state does not occur along a message, and e_j is a receive event, l_j must contain the send of an essential message to every neighbour of j. Let i be the processor whose

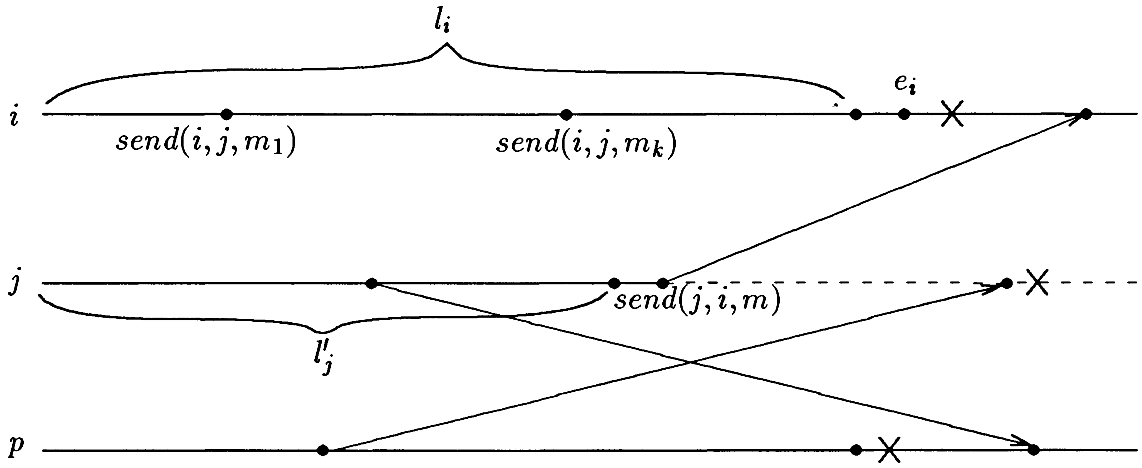


Figure 7.3: Proof of Theorem 3. No event in the dashed portion of j's local history can happen-before any event in any other processor's cut state.

first incidental or essential message from j in run r is sent last in l_j. Let $l_i \cdot e_i$ be i's cut state in r. (We know that e_i is a receive, but not of a message from j.) Let $send(j,i,m)$ be the first send of an incidental/essential message to i in l_j. Then neither $send(j,i,m)$ nor any event after it can happen-before any event in any other processor's cut state as follows. If $l_j{}'$ is the portion of j's cut state before $send(j,i,m)$, then $l_j{}' \cdot send(j,i,m)$ contains the send of an incidental/essential message to each of j's neighbours (see Figure 7.3). By assumption, no processor reaches its cut upon receiving a message from j, so each of these messages is received after the receiver's cut state. Thus any message sequence starting at or after $send(j,i,m)$ to an event of another processor's cut state will make the cut inconsistent. Therefore $send(j,i,m)$ does not happen-before any event in another processor's cut state, and no event which follows it in j's local history can do so either.

Let $m_1, \ldots, m_n$ be the messages sent from i to j in l_i that are not received in $l_j{}'$ (there is at least one, since by Lemma 9, i sends j an essential message in l_i; this essential message cannot be received in $l_j{}'$, as j's cut state only occurs at some point after $send(j, i, m)$). The above argument shows that $send(j, i, m) \not\rightarrow send(i, j, m_1)$. By an argument almost identical to that in Lemma 10, we can find a run r' in which every processor $p \neq j$ has the same cut state as in r, and $l_j{}' \cdot receive(j, i, m_1) \cdot \ldots \cdot receive(j, i, m_k)$ is j's cut state, for some $k \leq n$. (The only difference in the argument is the justification that there is no receive event in the intermediate partial run s^1 for which the corresponding send event is not also present. In the proof of Lemma 10, we invoke Lemma 8; here we use the argument in the last paragraph above.)

As in the proof of Lemma 9, if system sends from j to i have been disabled by the protocol in j's cut state in r', we can consider a possible run r'' and message m' such that: r'' has the same cut states as r', m' is the first message sent by j after its cut state, and there are no receive events between j's cut state and the sending of m' (see Figure 7.4). Since e_i is a receive event, it does not happen-before any event in j's cut state. There are no receive events between j's cut state and $send(j, i, m')$, so e_i does not happen-before $send(j, i, m')$ either. Therefore the hypotheses of Lemma 3 are satisfied, and there is a possible run in which message $l_i \cdot receive(i, j, m')$ is a prefix of i's cut state, so that m' is inconsistent with respect to the cut formed by the protocol. ∎

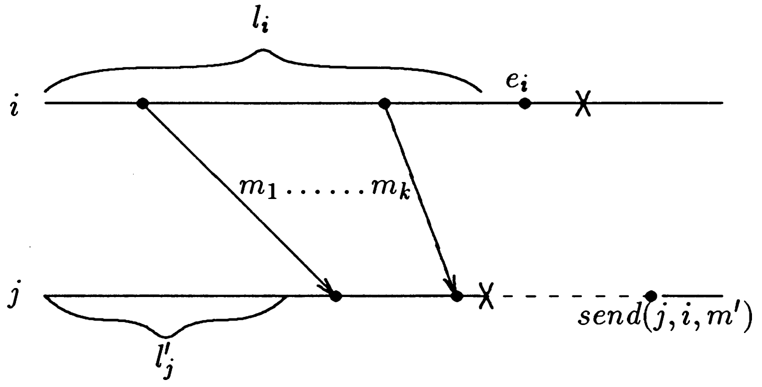


Figure 7.4: Proof of Theorem 3: the run r''. The dashed interval is receive-free.

7.4 Further Results

As we have stated, the proof of Theorem 2 is very similar to, but slightly simpler than, that of Theorem 3.

Proof of Theorem 2: We assume that there is a non-inhibitory consistent-cut protocol for FIFO systems, and use a sequence of lemmas almost identical to those used in the proof of Theorem 3. The lemmas differ only in this: rather than having to take into consideration the fact that the first send by a processor after its cut state might be a of protocol message, and occur only after a receive-free interval, we can assume that the first post-cut event is a system send to the appropriate processor. Because the protocol is non-inhibitory, this send event cannot be prevented. Thus the inconsistent message that will be produced in the final phase of the proof is guaranteed to be a system message. ∎

We now ask, as in Chapter 6, whether there is not some class of FIFO systems for which the impossibility results do not hold. In this case, there is indeed such a class.

Fact 1. There is a non-inhibitory consistent-cut protocol for the class of star networks (i.e. networks in which one processor has degree $N-1$ and all others have degree 1). Moreover, all messages will be consistent with respect to the designated cut.

Proof: For any such system, let I be the processor having degree $N-1$. The algorithm is trivial: I sends a Cut message to each processor $j \neq I$. I reaches its cut state upon sending the last such message; for $j \neq I$, j reaches its cut state upon the occurrence of the event $receive(j, I, Cut)$. This protocol is clearly non-inhibitory. We claim that the designated cut will always be consistent: for suppose there is some run r of the protocol in which this is not the case. Let m be an inconsistent message in run r; either the sender or the receiver of message m must be processor I. However, because the system is FIFO, any message sent from I to j after I's cut state will be received after the event $receive(j, I, Cut)$ and therefore after j's cut state. Thus I must be the recipient of the inconsistent message m. Now if $send(j, i, m)$ occurs after the cut and $receive(I, j, m)$ occurs before it, then $receive(I, j, m) \rightarrow send(I, j, Cut) \rightarrow receive(j, I, Cut) \rightarrow send(j, I, m) \rightarrow receive(I, j, m)$ and the happens-before partial order is violated. ∎

The same non-inhibitory protocol will of course also be successful for any class $\mathcal{K}$ which is a subset of the class of star networks. These, however, are the only exceptions.

Corollary 2. Let $\mathcal{K}$ be a class of FIFO systems $S = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M}, \mathcal{R})$, where membership in $\mathcal{K}$ is based solely on properties of the sets $\mathcal{I}$ and $\mathcal{C}$. Then the conclusions of Theorems 2 and 3 hold unless $\mathcal{K}$ is a subset of the class of all star-shaped networks.

Proof: The only characteristic network assumption that is required for the proofs of Theorems 2 and 3 is that there be two neighbouring processors each having degree at least two. In any class of systems which does not contain only star networks, there will be at least one system satisfying this property. All systems with an identical network configuration will also be included, and so our proofs follow through as above. ∎

7.5 Summary

In this chapter we have presented and proved theorems that together cover all the impossibility results for FIFO systems that we claimed in Chapter 4. The first demonstrates that no non-inhibitory CCP can exist, regardless of requirements on protocol message consistency. The second shows that, while local send inhibition suffices if protocol messages need not be consistent (as illustrated by *Flood*1), it is not sufficient to guarantee consistency of *all* messages. The conclusions of these two theorems carry over to all classes of FIFO systems with the sole exception of the subsets of the class of star networks.

Chapter 8

Message Complexity

In this chapter we look at the cost of achieving causal consistency in terms of the number of messages required by any consistent-cut protocol. This issue is not, in fact, separate from that of inhibition. As we will demonstrate, there is, for FIFO systems, a direct correlation between the degree—local or global—of inhibition used by a CCP, and the number of messages required. In particular, we show that any protocol which uses only local inhibition requires $O(|\mathcal{C}|)$ messages. On the other hand, if a consistent-cut protocol for FIFO systems use global inhibition, then it need use only $O(N)$ messages; in the worst case, the exact lower bound is $2N-3$ messages. Thus there is a trade-off between the degree of inhibition and the message complexity. The quantity $2N-3$ is a lower bound for a CCP that is required to work for all FIFO systems. There are classes of systems, however, for which there are CCPs which use fewer messages. For instance, there are consistent-cut protocols using only $N-1$ messages which are successful for all FIFO systems in which the processor network is a tree.

In the non-FIFO case, global inhibition is necessary to any consistent-cut protocol. As in the FIFO case, a globally inhibitory protocol requires $O(N)$ messages, $2N-2$ to be precise. However, unlike for FIFO systems, this bound holds for all classes of non-FIFO systems.

Our results assume that in each run of the protocol there is a distinguished initiator, and a first protocol event that causally preceded all other protocol events. Note that this is the case in the three protocols of Chapter 3. One could imagine a protocol which did not have this property; for instance, one could construct a protocol resembling *Tree* of Chapter 3 but in which the *PrepareCut* phase of messages was omitted. In this protocol, each leaf of the network spanning tree would eventually have to start its protocol events spontaneously and independently. Given the potential time lags between the points at which each leaf processor chose to start sending *Cut* messages, such a protocol seems of limited practical use or interest.

We start our chapter with a brief discussion of the ideas and techniques that we will use for the protocols and lower bound proofs for both non-FIFO and FIFO systems. We then turn our attention non-FIFO systems first. We present a consistent-cut protocol that uses $2N-2$ messages and argue that it is indeed successful for all non-FIFO systems. This argument is followed by our proof that $2N-2$ is indeed a lower bound on the message complexity of CCPs for non-FIFO systems. For FIFO systems, we look first at the case of global inhibition. We present a globally inhibitory CCP for FIFO systems which requires $2N-3$ messages in the worst case. We argue that the protocol is correct, and that any other would require at least as many messages. We then turn our attention to CCPs which use local inhibition only, and prove that any such protocol requires

$O(|\mathcal{C}|)$ messages. We close the chapter with a discussion of precise lower bounds.

8.1 Results for Non-FIFO Systems

8.1.1 Preliminaries

We start by making precise an idea that has played a part in the non-FIFO correctness and impossibility proofs which have appeared in previous chapters, and which plays a key role in our arguments below.

Lemma 14. Let S be any system, P be a consistent-cut protocol, and r be a run of $P(S)$. Let i and j be any pair of neighbouring processors. Let l_i be the last proper prefix of i's cut state in run r in which receives are enabled, and let e_i be the event following l_i. Let l'_i be the first extension of i's cut state in run r in which sends to j are enabled, and let e'_i be the event following l'_i. Define l_j,e_j,l'_j, and e'_j similarly with respect to i. Then (1) if e'_i is $send(i,j,m)$ for some message m, it must be the case that $e_j \rightarrow e'_i$; and (2) if e'_j is $send(j,i,m')$ for some message m', it must be the case that $e_i \rightarrow e'_j$.

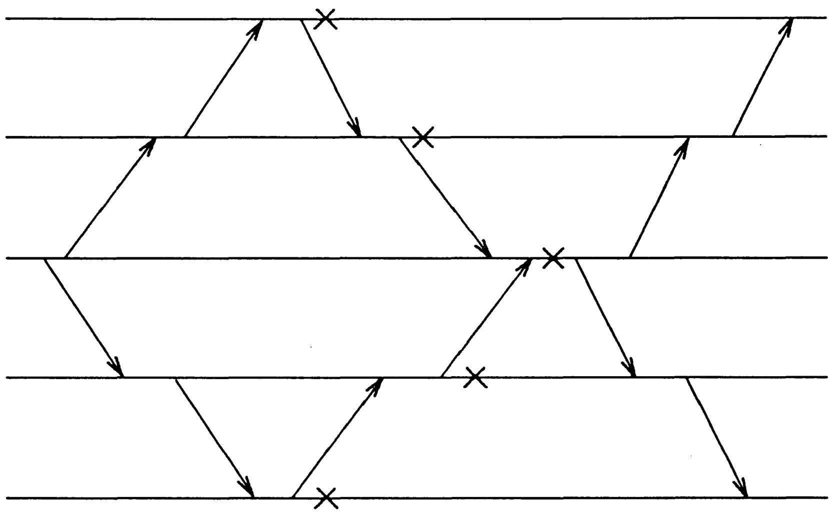
Proof: Suppose that $e'_i \equiv send(i,j,m)$ but $e_j \not\rightarrow e'_i$. Then by Lemma 2 of Chapter 5, there exists another protocol run in which $l'_i \cdot e'_i$ is a prefix of i's local history and $l_j \cdot receive(j,i,m)$ is a prefix of j's local history. In this run, j's cut state cannot occur until after the event $receive(j,i,m)$ and we have an inconsistent message, and a contradiction. Thus $e_j \rightarrow e'_i$. The argument for the second conclusion of the lemma proceeds identically. ∎

In effect, the lemma states that in any run of a CCP, and for any pair of neighbours i and j, the last pre-cut event of i that could be a receive from j must happen-before the first post-cut event of j that could be a send to i. We proved protocol *Tree* correct in Chapter 3 by showing that this property holds in every protocol run. In an attempt to determine whether or not the $3(N-1)$ messages of protocol *Tree* are necessary to guarantee that this property hold, we look at the protocol a bit more closely.

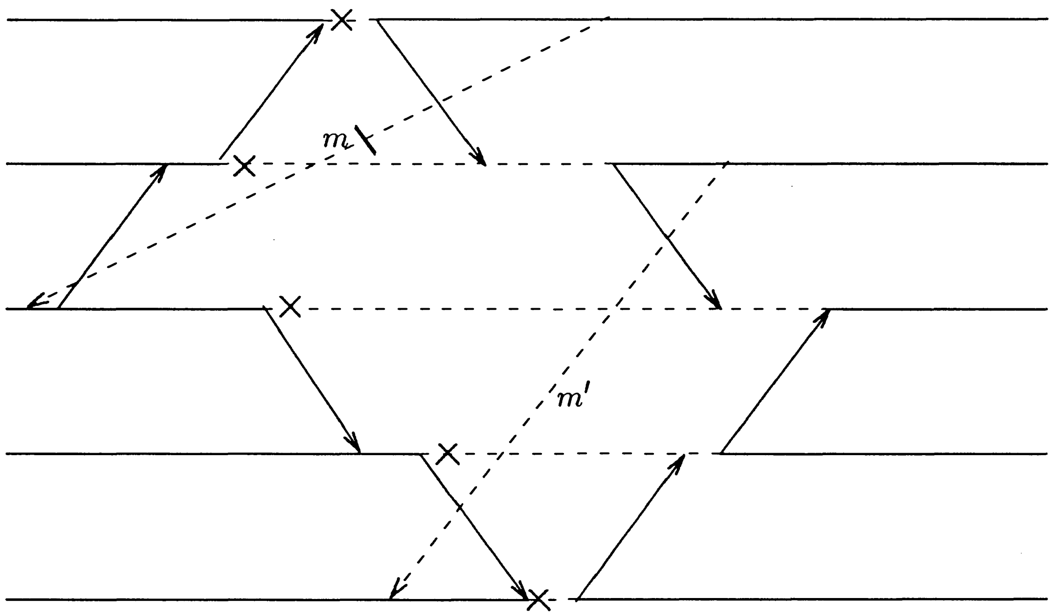
Why are three phases of messages necessary? What would happen if processors reached their cut state after forwarding the first-phase message, and then disabled system sends until receiving a second-phase message from each child? (See Figure 8.1.) Upon examination, one can see that this would be sufficient to guarantee the consistency of any message between a pair of processors where one is an ancestor of the other in the spanning tree, provided that receives were disabled between each processor's first protocol event and its cut state. (See Figure 8.1 (b). The inconsistent message m would produce circular causality.) However, it would still be possible to have an inconsistent message between neighbours where neither is an ancestor of the other (for example, the message m' in Figure 8.1 (b)). The question then arises: for any given system, is there a spanning tree of the processor network with the property that any neighbouring processors share an ancestor/descendant relationship in that tree?

Lemma 15. In any connected graph G there is a spanning tree T such that for any pair of nodes u and v, if u and v are neighbours in G then either u is an ancestor of v in T or vice-versa.

Proof: Such a tree can be built using a depth-first search of the graph. We start with an arbitrary node v_1 and set $V' = \phi$. At each step, we look at all of the neighbours of the last node v_n to have been added to V'. If the node v_n has neighbours which have not yet been added to V', then we arbitrarily choose



(a)



(b)

Figure 8.1: The protocol *Tree* (a) and the proposed modification (b).

a v_{n+1} from amongst them. If there are no such neighbours, then we back up the current tree to the first of v_n's ancestors which does have such neighbours, and choose v_{n+1} from amongst them. In either case, the node v_{n+1} is added to V' and the edge (v_n, v_{n+1}) to T. We continue this procedure until all nodes have been added to V', which will eventually be the case since G is connected.

We claim that T satisfies the requirements of the lemma. For suppose that there exist nodes u and v which are neighbours in G and such that neither is an ancestor of the other in T. Of u and v, let u be the node which is added to V' earlier in the construction. Let v' be the last node that is added to u's subtree in the construction of T. It must be the case that every node added to V' between the addition of u and the addition of v' is a member of u's subtree. By hypothesis, v is not in u's subtree. But $v \in V'$ at the completion of u's subtree, or it would be added to V' as a child of u. This contradicts our assumption that v is not a descendant of u in T. ∎

We refer to a tree which has the property in the statement of Lemma 15 as a *depth-first-search* tree. These trees will prove to be instrumental in the construction of minimum-complexity consistent-cut protocols.

8.1.2 A Minimizing CCP for Non-FIFO Systems

Using a depth-first-search spanning tree, we have for non-FIFO systems the consistent-cut protocol *Minimize*1, described in Figure 8.2.

Proof of correctness of protocol *Minimize1*: Suppose that in some run r of Protocol 8.2 there is an inconsistent message m from processor i to processor j. Because of the form of the spanning tree T, it must either be the case that i is an ancestor of j in T or vice versa. Suppose first that j is an ancestor of i. Let

- Initiator I:
  - start CCP; disable all receive events;
  - send to each processor in *children*(I) a *Cut* message; (**Cut**)
  - disable all system send events; re-enable all disabled receive events.
- Internal node i upon reception of a *Cut* message:
  - disable all receive events;
  - send to each processor in *children*(i) a *Cut* message; (**Cut**)
  - disable all system send events; re-enable all disabled receive events.
- Leaf node i upon reception of a *Cut* message:
  - (**Cut**) disable all system sends;
  - $send(i, parent(i), Resume)$;
  - re-enable system sends.
- Internal node i, having received *Resume* messages from all *children*(i):
  - $send(i, parent(i), Resume)$;
  - re-enable system sends.
- Initiator I, having received *Resume* messages from all *children*(I):
  - re-enable system sends.

Figure 8.2: **Minimize1** T is a depth-first-search spanning tree of the network rooted at I; *parent*(i) and *children*(i) are the parent and children of i in T.

k be the child of j which is also either an ancestor of i or i itself. Then, since there are no receive events between $send(j,k,Cut)$ and the end of j's cut state, we have

$$send(j,k,Cut) \rightarrow receive(i,parent(i),Cut) \rightarrow send(i,j,m) \rightarrow$$

$$receive(j,i,m) \rightarrow send(j,k,Cut).$$

Thus an inconsistent message from i to its ancestor j would produce causal circularity. Therefore it must be that i is an ancestor of j.

The processor i cannot send any messages after its cut state until it has received $Resume$ messages from all of its children in T. Let k be the child of i which is also either an ancestor of j or j itself. Then

$$send(j,parent(j),Resume) \rightarrow receive(i,k,Resume) \rightarrow send(i,j,m) \rightarrow$$

$$receive(j,i,m) \rightarrow send(j,parent(j),Resume)$$

and once again we have causal circularity. Therefore there cannot be an inconsistent message in any run r of protocol $Minimize1$. ∎

Note that $Minimize1$ uses $2N-2$ messages in any system, regardless of network topology, or of the shape of the spanning tree itself. We prove in the following subsection that $2N-2$ is a lower bound on the message complexity of CCPs for non-FIFO systems.

8.1.3 Lower Bound Theorem for Non-FIFO Systems

Theorem 4. The lower bound on the message complexity of consistent-cut protocols for non-FIFO systems is $2N-2$.

In order to prove our theorem, we need to prove that for any CCP for non-FIFO systems, there is some protocol run in which $2N-2$ protocol messages are required to guarantee the consistency of the designated cut. We can therefore make any worst-case assumptions about enabling relations on sequences of system events, and about the occurrence of system events outside of any intervals in which they are inhibited by the protocol.

Let P be a consistent-cut protocol for non-FIFO systems, and let r be a run of the protocol with respect to some system S. We can assume that no system message is sent in r until after the completion of the protocol (i.e. there is no system send event which happens-before any protocol event, or which is included in a local state in which events are still disabled by the protocol). We can also assume that a series of system sends to i's neighbours are the first post-cut system events of each processor i. (These sends may occur only after an interval of protocol events during which system sends are inhibited by the protocol.) Now, by Lemma 14 there must, in run r, be causal paths, in both directions, between every pair of neighbouring processors. Moreover, all of the messages on these paths must be protocol messages. We assume that the protocol uses the minimum number of messages necessary, and show that this number must be at least $2N-2$.

Firstly, let I be the initiator of the protocol in run r. The event *Start CCP* occurs at I, and *Start CCP* happens-before every other protocol event in run r. Because the processor network is connected, and because there are paths of protocol messages linking every pair of neighbours, there must be message paths from the initiator to every other processor in the network. For each processor other than the initiator, consider the first message received by that processor in the run r. There are $N-1$ such messages, which we refer to as *original messages.* If G is a directed graph where the nodes correspond to the processors in S, and the edges correspond to messages sent in r, then the edges corresponding to the original messages together form a spanning tree T, rooted at I, of G.

If i is j's parent in the tree T, then clearly there must be a channel between them in the processor network. The original message from i to j could enforce the necessary causal precedence in the $i-j$ direction; i.e. this message may ensure that the last event before i's cut state which could be a receive from j happens-before the first post-cut system send from j to i. However, we must still show that the corresponding $j-i$ causal relationship also holds.

Lemma 16. Every internal processor in T must receive a message in addition to the original one.

Proof: To guarantee that there is an appropriate causal precedence between a processor i and i's parent in T, there must be a message path from i to $parent(i)$. The first message m in this path must be sent after the reception of the original message m_0 to i, as $receive(i, parent(i), m_0)$ is the first protocol event occurring at i. Therefore, the path from i to $parent(i)$ cannot use the original message m into $parent(i)$: if it does, then

$$receive(i, parent(i), m_0) \rightarrow send(i, q', m') \rightarrow \ldots \rightarrow receive(parent(i), q, m) \rightarrow$$

$$send(parent(i), i, m_0) \rightarrow receive(i, parent(i), m_0)$$

and there is causal circularity in run r, which is impossible. Therefore $parent(i)$ must receive some message other than the original one. ∎

We have thus shown that, in addition to the $N-1$ original messages, there must be at least as many more messages received as the number N_{int} of processors which are internal in T. We also know that, in addition to the original messages, there must be as many more sent as the number N_{leaf} of processor which are leaves in T: each leaf must send one, otherwise there could not be a path from it to its parent. Unfortunately, some of the additional received messages will of course be the same as some of the additional sent messages, and so we have not yet proved that there must be $N-1$ messages in addition to the $N-1$ originals.

In order to prove that there must indeed be $N-1$ additional messages we look at the messages received by each processor in the network. We know that each internal processor must receive an additional message; if it receives more than one, we refer to all but one as *extra* messages. Any additional message received by a leaf processor we also refer to as extra. Our object is to show that if there are N_{leaf} leaves in T, then there are at least $N_{leaf}-1$ extra messages received in r; then we will have shown the presence of $N_{int}+N_{leaf}-1=N-1$ additional messages in r.

We proceed by showing that if, in run r, there are message paths to a single processor p from processors in n different branches in T, then those paths must contain at least n extra messages if p is a leaf in T, and at least $n-1$ extra messages otherwise. We start with the case where p is a leaf.

Lemma 17. Let p be a leaf processor in the tree of original protocol messages T. Suppose that there are, in r, paths to p from n processors $p_1, ..., p_n$ where each p_i is a leaf in T. Then there must be at least n extra messages on these paths.

Proof: Let m_i be the first message in a p_i to p path (there may be more than one such path); m_i is not an original message, since p_i is a leaf in T. Let $q_1, ..., q_n$ be the (not necessarily distinct) processors which receive these messages.

How could m_i *not* be extra? It would require that q_i be an internal node, and that all non-original messages to q_i be included amongst $m_1, ..., m_n$. Suppose that there are n_1 such q_i's. Then there are $n - n_1$ extra messages among $m_1, ..., m_n$ (all but one message to each of the n_1 q_i's are extra).

We claim that each of the n_1 q_i's must send an additional message; i.e. there must be a path from q_i to p whose first edge is non-original. For let c be the child of q_i whose original message m from q_i is the last original message sent by q_i. The path from c to q_i must use (end with) one of $m_1, ..., m_n$, since these include all non-original messages to q_i. This message m_j must be received after all original messages from q have been sent. However, m_j is also a message in a path from p_j to p. Therefore the path must continue on the other side of q_i. (See Figure 8.3; straight arrows are direct messages, curved arrows represent message paths.) The path cannot continue through an original message since m is the last original message sent by q_i; thus q_i sends an additional message. Hence there are n_1 additional messages from the q_i's in the paths to p. Again, all of these are extra unless there is an internal processor q which receives no non-original messages other than those sent in the first two "rounds" of messages. If there are n_2 such processors, then there will be $n_1 - n_2$ messages which are not extra. We argue as above that all n_2 such processors must again send an additional message each, and repeat. If k is the length of the longest path from any p_i to p, then we will have $(n - n_1) + (n_1 - n_2) + (n_3 - n_2) + ... + (n_{k-1} - n_k) + n_k = n$ extra messages in total. ∎

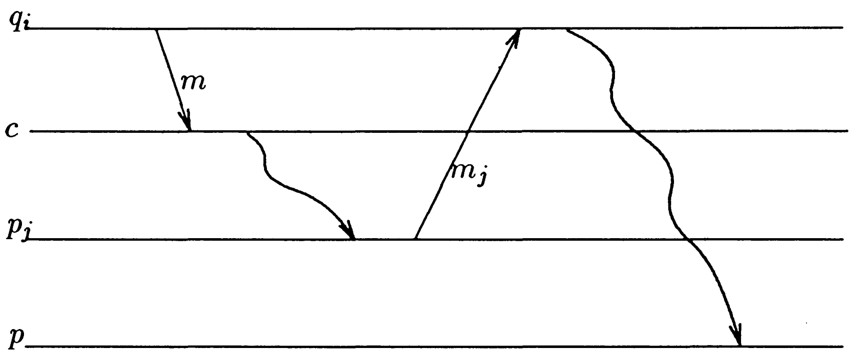


Figure 8.3: The q_i to c message is the last original message sent by q_i; the message m_j lies on both the c to q_i and p_j to p paths.

The case where p is an internal node is argued in a similar manner.

Lemma 18. Let p be an internal processor in the tree of original protocol messages. Suppose there are paths to p from n processors $p_1, ..., p_n$ such that each p_i is a leaf in the tree of original messages. Then there must be $n-1$ extra messages on these paths.

Proof: We proceed as in the proof of Lemma 17, except that at each stage, one of the n_j q_i's could be p itself, in which case no additional message need be sent. This can occur at at most one stage, since all non-original messages to p must be included among the messages already considered at that stage. Therefore it is not possible that p be a receiving processor for any message sent in a subsequent stage. ∎

Lemma 17 and Lemma 18 give us the minimum number of extra messages that must be present in paths such as those described. This minimum was determined by looking at the number of processors that *must* send non-original messages. In fact, other processors on the paths may also send such messages, resulting in a higher count of extra messages, as explained below.

Lemma 19. Let p be a leaf processor in the tree of original protocol messages. Suppose that there are, in run r, message paths to p from n processors $p_1, ..., p_n$ where each p_i is a leaf in T. Let M be the set of all messages appearing in these paths. Suppose there are n' processors q such that q receives a non-original message $m \notin M$, and q sends a non-original message $m' \in M$. Then there are at least $n + n'$ extra messages in the paths.

Proof: The number of extra messages on the paths is the total number of non-original messages, minus the number of processors all of whose incoming non-original edges are included in that total. If there are n' more non-original edges in the paths than those counted in the proof of Lemma 17, then there are n' more extra. ∎

Lemma 20. Let p be an internal processor in the tree of original protocol messages. Suppose that there are, in run r, message paths to p from n processors $p_1, ..., p_n$ where each p_i is a leaf in T. Let M be the set of all messages appearing in these paths. Suppose there are n' processors q such that q receives a non-original message $m \notin M$, and q sends a non-original message $m' \in M$. Then there are at least $n + n' - 1$ extra messages in the paths.

Proof: As above.

We call a protocol message m from processor q to processor p a *terminal* message in the run if there is no processor p' and protocol message m' such that

$receive(p, q, m) \rightarrow send(p, p', m')$. Note that no original message is terminal. Also, since the number of messages involved in the run has been assumed to be the minimum required by the protocol then there will be no terminal message into a leaf processor (such a message could not be necessary to any child-parent path).

Suppose there are k terminal messages $m_1, ..., m_k$ in r, where m_i is a message from p'_i to p_i for $i = 1, ..., k$. For each i we consider the message paths in r which end in m_i; i.e. all those messages whose receptions happen-before $send(p'_i, p_i, m_i)$. We want to count the leaves of T and the extra messages on these paths; however, this task is rendered difficult by the fact that some of these may be duplicated. (See Figures 8.4 and 8.5. The first shows a hypothetical configuration of protocol messages in a run; the second illustrates the paths ending in the terminal messages m_1 and m_2.) To ensure that our count is accurate, we perform the following pruning operations. For each processor p in the system, let m^p be the last message sent by p that is contained in any of these paths. For any message m which appears in these paths, $m \neq m^p$ for any p, and for each path in which m occurs, delete from the path all messages leading up to m. That is, delete all messages m' such that the reception of m' happens-before the sending of m. If there is an original message with no non-original messages leading up to it, then delete that as well. (Figure 8.6 illustrates the effect of the pruning operations on the message paths of Figure 8.5.) For each i let T_i be the tree consisting of what is left of the paths through m_i. The T_i's now have the following properties, which will allow us to count messages precisely.

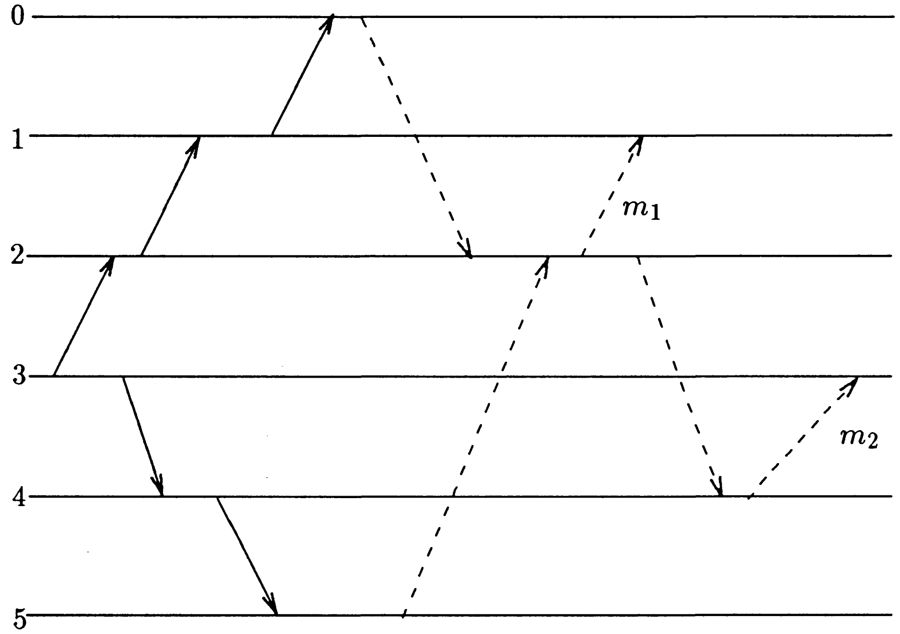


Figure 8.4: A hypothetical run. The solid arrows indicate original messages and dotted arrows additional messages.

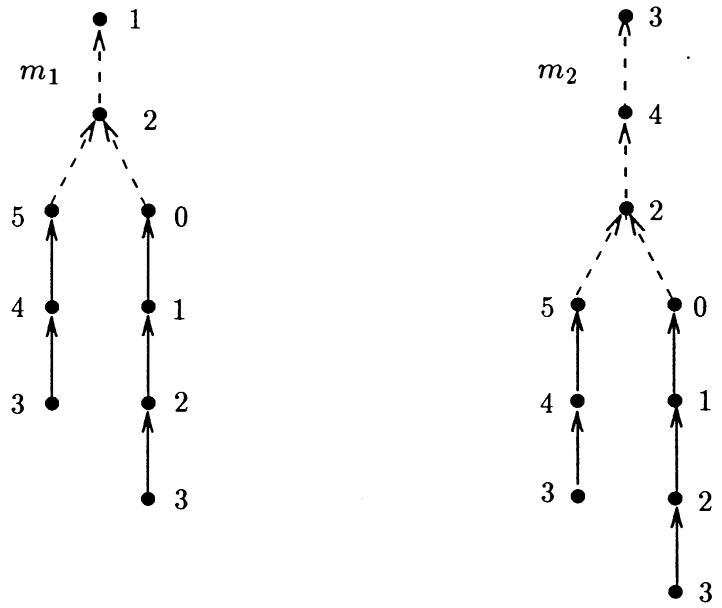


Figure 8.5: The messages leading up to m_1 and m_2.

Figure 8.6: The pruned message paths to m_1 and m_2.

Lemma 21. Every non-original message in r has a corresponding edge in exactly one of the T_i's.

Proof: Every non-original message must have appeared in one of the paths before the pruning; none of these messages can have been completely excluded after it. For suppose m is a non-original message from processor q^0 to processor q^1. Let m^1 be the last message sent by q^1 and q^2 be the processor to which it is sent, m^2 be the last message sent by q^2 and q^3 be the processor to which it is sent, and so on to $m^j \equiv m_i$ for some i. None of these messages is deleted under the pruning instructions, and so m will appear in T_i. No message can appear twice in the T_i's: each m_i appears only once since $receive(p_i, p_i', m_i)$ does not happen-before any protocol send event and hence does not appear in T_j for any $j \neq i$; each other message m will appear only in the same T_i as the last protocol message that is sent by the processor which receives m. ∎

Lemma 22. If processor p receives a message in more than one T_j, then p is p_i for some i, i.e. p receives a terminal message m_i.

Proof: Suppose p is not one of the p_i's. Then all messages received by p are received before p sends its last message m and will thus be included in the same T_j as m. By Lemma 21, no message appears in more than one of these pruned trees, and so p will receive messages in that single T_j only. ∎

Lemma 23. No T_i can be processor-disjoint from each other T_j. Moreover, the T_i's cannot be partitioned into groups such that each T_i belongs to some group, but one group is processor-disjoint from all the others.

Proof: All the non-original edges are present in the T_i's, and there must be paths from every child in the tree of original messages to its parent in that tree. Therefore if T_i has no processor in common with any other T_j, T_i must either be empty or else contain all processors. The argument in the case of groups of T_i's is identical. ∎

Recall that T is the tree of original messages in run r, with N_{int} internal nodes and N_{leaf} leaves. We know that there must be N_{int} messages in addition to those of T, and are trying to prove that there must be $N_{leaf} - 1$ extra messages, for a total of $2N - 2$ messages. Having established the properties of the T_i's, we now endeavour to count the extra messages contained amongst them. We present two final lemmas which will aid us to accomplish this task.

Lemma 24. For each $i = 1, ..., k$, if $n_i > 0$ is the number of processors contained in T_i which are leaves in the tree T of original messages, then T_i contains $n_i - 1$ extra messages, and these messages *do not include* m_i.

Proof: Recall that each message m_i is sent from processor p_i' to processor p_i, and that no p_i is a leaf in T. Thus the n_i leaves of T which are included in T_i lie on the paths leading up to p_i'; they may include p_i' itself. In either case, Lemmas 18 and 17 guarantee that there will be $n_i - 1$ extra messages leading up to p_i', i.e. m_i is not included in the count. By Lemma 21, the edge sets of the T_i's are disjoint, so no edge can have been counted as extra twice. Also, by Lemma 22, the only processors which may receive messages in more than one T_i are the p_i's, and so we cannot have counted all the non-original edges into some processor as extra. ∎

Lemma 25. Suppose there are terminal messages $p_{i_1}, \ldots, p_{i_K}$ which are all received by the same processor p. For $j = 1, ..., K$, let n_{i_j} be the number of leaves of T which appear in T_{i_j}. Then there are $n_{i_1} + \ldots + n_{i_K} - 1$ extra messages contained in $T_{i_1} \cup \ldots \cup T_{i_K}$.

Proof: By Lemma 24, each T_{i_j} contains $n_{i_j} - 1$ (or 0, if $n_{i_j} = 0$) extra messages, with no m_{i_j} being counted as extra. Since each m_{i_j} is received by the same processor p, $K - 1$ of these can also be counted as extra. As a result, there are at least $n_{i_1} + \ldots + n_{i_K} - K + (K - 1)$ or $n_{i_1} + \ldots + n_{i_K} - 1$ extra messages in $T_{i_1} \cup \ldots \cup T_{i_K}$. ∎

Proof of Theorem 4: Suppose that $k = 1$; i.e. there is only one terminal message m_1 in run r, where m_1 is sent by p_1' and received by p_1. Then there are paths from each of T's leaf processors to p_1 which end with the message m_1, since by Lemma 21, all non-original messages (and hence all leaves of T) will appear in the paths through m_1. Thus if there is only one terminal message in r we are done: there will be $N_{leaf} - 1$ extra messages on the paths to p_1.

If $k > 1$, then for each i, let T_i consist of the message paths into m_i constructed as described above, and let L_i be the set of leaves of T which appear in T_i. We can assume without loss of generality (by Lemma 25 and the second part of Lemma 23) that the processors $p_1, ..., p_k$ which receive the terminal messages $m_1, ..., m_k$ are all distinct; we can also assume that each T_i contains at least one leaf of T. We set $i_1 = 1$ and pick an i_2 such that T_{i_1} and T_{i_2} are not processor-disjoint; by Lemma 23 it is possible to find such an i_2. By Lemma 22, there are four possible relationships between the processor set of T_{i_1} and that of T_{i_2}:

1. some processor p in T_{i_2} sends (but does not receive) a message in T_{i_1};

2. some processor p in T_{i_1} sends (but does not receive) a message in T_{i_2};

3. p_{i_2} appears in T_{i_1};

4. p_{i_1} appears in T_{i_2}.

(Note that these possibilities are not mutually exclusive.)

We set $n_1 = |L_{i_1}|$, and $n_2 = |L_{i_2} - L_{i_1}|$: this latter quantity is the number of leaves which appear in T_{i_2} but not in T_{i_1}. We know that there must be at least $n_1 - 1$ extra messages in T_{i_1} and at least $n_2 - 1$ extra messages in T_{i_2}. We wish to show that there are in fact at least $n_1 + n_2 - 1$ extra messages altogether. (To be scrupulously accurate, we remark that if one or both of n_1 and n_2 is 0, then any quantity appearing in this argument which might be negative (for example, $n_1 - 1$) should be replaced by the number 0.)

Suppose (1) holds, i.e. there is some processor p in T_{i_2} which sends (but does not receive) a message in T_{i_1}. If p is a leaf in the tree of original messages T, then n_2 is actually one less than the number of T's leaves which appear in T_{i_2}. Therefore there are at least n_2 extra messages in T_{i_2}, and at least $n_1 + n_2 - 1$ altogether. Suppose p is an internal node in T. It appears (by assumption) in T_{i_1}; however, not all the non-original messages received by p are included in T_{i_1} (none are). It is not possible that the message from p which is included in T_{i_1} be original, or it would have been deleted in the construction of T_{i_1}. Therefore by Lemmas 19 and 20, there is actually one more extra message included in T_{i_1}, and hence there are at least $n_1 + n_2 - 1$ extra messages altogether.

If (1) does not hold but (2) does, then we use a symmetric argument to show, as above, that there are at least n_1+n_2-1 extra messages in T_{i_1} and T_{i_2} together.

Suppose that (3) is the first of the four cases to apply: p_{i_2} appears in T_{i_1}. There is a non-original message (m_{i_2}) which is received by p_{i_2} but which is not included in T_{i_1}. If the message sent by p_{i_2} in T_{i_1} is non-original, then (by Lemmas 19 and 20) there are at least n_1 extra messages in T_{i_1} and we are done. We claim that the message m sent by p_{i_2} in T_{i_1} *must* be non-original. For suppose otherwise. Let c be the child of p_{i_2} which receives m. The message m would have been pruned out of T_{i_1} if it were not the last message sent by p_{i_2}. We know that there must be a message path from c to p_{i_2}; since m is the last message sent by p_{i_2}, this path must end in a terminal message to p_{i_2}. However, c does not appear in T_{i_2} (or we would have fallen into case (2) above), and we have assumed that the processors which receive terminal messages are distinct. This leads us to a contradiction.

Case (4) is argued symmetrically to case (3).

We continue in this fashion: at the jth stage, we choose $T_{i_{j+1}}$ which is not processor-disjoint from $T_{i_1} \cup ... \cup T_{i_j}$. The argument above is repeated, with $T_{i_1} \cup ... \cup T_{i_j}$ in the place of T_{i_1}, $T_{i_{j+1}}$ in place of T_{i_2}, etc.; thus we show that there are at least $n_1 + ... + n_{j+1} - 1$ extra messages contained in $T_{i_1} \cup ... \cup T_{i_{j+1}}$. Finally, we have $n_1 + ... + n_k = N_{leaf}$, and $n_1 + ... + n_k - 1 = N_{leaf} - 1$ extra messages, as desired. Our proof is now complete. ∎

8.2 Results for FIFO Systems

In this section we look at lower bounds on the message complexity of consistent-cut protocols for FIFO systems. We consider bounds on both locally and globally inhibitory protocols. We start by examining the latter, as the protocols and arguments involved closely resemble those for non-FIFO systems contained in the preceding sections.

8.2.1 Globally Inhibitory Protocols

The ideas behind the globally inhibitory protocol we present for FIFO systems are much the same as those we discussed in the context of non-FIFO systems in the first section of this chapter. Once again we use a depth-first-search spanning tree to minimize communication, and again we are constrained by the necessity of having causal paths between neighbouring processors. There is, however, one major difference: a single message m from processor i to processor j may be sufficient to prevent an inconsistent message between i and j in either direction. It may prevent an inconsistent $j-i$ message as in the non-FIFO case; in addition, if receive events from i are disabled between the reception of m and the end of j's cut state, then there can be no inconsistent $i-j$ message either, due to the FIFO condition. Thus in protocol $Minimize2$ below, a processor need not disable system sends to its children, and no processor need send a *Resume* message to its parent. $Minimize2$ uses global send inhibition and local receive inhibition, is a consistent-cut protocol for FIFO systems which guarantees consistency of all messages.

Proof of correctness of protocol *Minimize2*: Suppose that in some run r of Protocol $Minimize2$ there is an inconsistent message m from processor i to processor j. Because of the form of the spanning tree T, it must either be the case that i is an ancestor of j in T or vice versa. Suppose first that j is an ancestor of i. Let k be the child of j which is also either an ancestor of i or i itself. Then, since there are no receive events between $send(j,k,Cut)$ and the end of j's cut state, we have $send(j,k,Cut) \rightarrow receive(i,parent(i),Cut) \rightarrow send(i,j,m) \rightarrow receive(j,i,m) \rightarrow send(j,k,Cut)$.

- Initiator I:
  - start CCP;
  - send to each processor in *children*(I) a *Cut* message; **(Cut)**
  - disable system sends to all processors other than those in *children*(I).
- Internal node i, upon reception of a *Cut* message:
  - disable all receive events;
  - send *Cut* messages to *children*(i); **(Cut)**
  - re-enable the disabled receive events;
  - disable system sends to all processors other than those in *children*(i);
  - for each k such that k is one of i's ancestors other than *parent*(i): if k is not a neighbour of one of i's descendants, then *send*(i, k, Resume).
- Leaf nodes i, upon reception of a *Cut* message:
  - **(Cut)**; for each k such that k is one of i's ancestors other than *parent*(i): if k is a neighbour of i, then *send*(i, k, Resume);
- Internal node or initiator i, upon the event *receive*(i, j, Resume):
  - re-enable sends to all ancestors of j in T.

Figure 8.7: **Minimize2** T is a depth-first-search spanning tree of the network, rooted at I; *parent*(i) and *children*(i) are the parent and children of i in T.

Since there cannot be causal circularity in a run, there cannot be an inconsistent message from processor i to its ancestor j.

Suppose then that j is a descendant of i. If i is j's parent in T, then j reaches its cut state after receiving a *Cut* message from i and before receiving any other messages from i. Any message sent by i after its cut state is sent after the event $send(i, j, Cut)$ and hence will be received at j after the event $receive(j, i, Cut)$. Thus it will be received after j's cut state and cannot be inconsistent. So i and j cannot be parent and child. In this case, in accordance with the protocol, i suspends system sends to j upon reaching its cut state. Either j or one of its descendants in T must send a *Resume* message to i, and i must receive this message before performing any post-cut system sends to j. Let k be the processor (either j or one of its descendants) which sends the *Resume* message to i. Then

$$send(k, i, Resume) \rightarrow receive(i, k, resume) \rightarrow send(i, j, m) \rightarrow$$

$$receive(j, i, m) \rightarrow send(k, i, Resume)$$

and we have causal circularity. Therefore it is impossible that there be an inconsistent message from any processor i to any processor j. ∎

If the spanning tree used by $Minimize2$ has K leaves, then the protocol requires at most $2N - 1 - 2K$ messages, since neither a leaf, nor an internal node whose children are all leaves, need receive any message besides the original one. Thus the maximum number of messages that Protocol 8.7 can use in any FIFO system is $2N - 3$; this occurs when every internal node of the depth-first-search spanning tree has only one child. This is the case, for example, in any system for which the channel set C is completely connected or in which there is a Hamilton circuit. We claim that $2N-3$ is the least number of messages that any CCP can use under those circumstances.

8.2.2 Lower Bound Theorem for Globally Inhibitory Protocols

Theorem 5. Any consistent-cut protocol for FIFO systems requires at least $2N-3$ messages in the worst case.

Proof: We consider a system S where the processor network is completely connected, and a run r of the protocol. Let T be the spanning tree of the network that is created by the set of messages m_i such that m_i is the first protocol message received by processor i. Suppose that there are two processors i and j such that i is not *parent*(j), j is not *parent*(i), and neither receives an additional message. Since neither processor sends a message directly to the other, there must be a message path from i to j, and also from j to i The message path from i to j must use the original *parent*(j) to j message m_j, and the path from j to i must use the original *parent*(i) to i message m_i. Then

$$send(parent(i), i, m_i) \rightarrow receive(i, parent(i), m_i) \rightarrow send(i, k, m) \rightarrow$$

$$send(parent(j), j, m_j) \rightarrow receive(j, parent(j), m_j) \rightarrow$$

$$send(j, k', m') \rightarrow send(parent(i), i, m_i)$$

and there is causal circularity in the run. Therefore there can be at most two processors—one the parent of the other in the tree of original messages—that do not receive an additional message. There are thus $N-1$ original messages and at least $N-2$ additional messages, for a total of at least $2N-3$ messages altogether.■

Note that this proof is much simpler than that given for the lower bound for non-FIFO protocols. Using the same type of argument as that here (i.e. by looking at the case of a completely connected network), we could easily have shown that, in the worst case, $2N - 2$ messages are required. We chose to prove the stronger result of Theorem 4, which gives a lower bound of $2N - 2$ without relying on the features of the processor network topology, and hence holds for any class of non-FIFO systems.

8.2.3 Lower Bound Theorem for Locally Inhibitory Protocols

In our final section we turn our attention to locally inhibitory CCPs for FIFO systems. We first prove a general lower bound theorem for these CCPs, and then discuss exact bounds.

Theorem 6. Any locally inhibitory consistent-cut protocol for FIFO systems uses $O(|\mathcal{C}|)$ messages.

Proof: Let P be a consistent-cut protocol which uses local inhibition (of send events, receive events, or both) only. It is possible that, in a given run, and for some processor, the protocol inhibits receive events for some interval immediately preceding that processor's cut state. However, since the protocol uses local inhibition only, there is some run where no communication is required to re-enable any receive events that may have been disabled; thus there will be no receive events at any processor between the point at which any receive events are first disabled, and the point at which they are re-enabled. Similarly, in this particular run there may be send events of some processor which are inhibited by the protocol for some interval immediately following that processor's cut state. Again, there must be some other run in which these events are re-enabled with no intervening receives. It is this last run which we wish to examine.

Let r be a run of the protocol in which, for each processor i, there are no receive events between the first pre-cut state in which there are events disabled by the protocol, and the first post-cut state after which no events remain disabled. We may assume that the first system events following a processor's cut state are a series of sends to its neighbours, and that these are the first system sends occurring at that processor in run r. Suppose that i and j are neighbours such that i does not send a message to j in r, and vice versa. Let l_i be the lst proper prefix of i's cut state in which receives are not disabled by the protocol; we can assume that receives from j are enabled at this point. Let e_i be the event occurring in l_i. Define l_j and e_j similarly for j. Finally, let $send(i,j,m)$ be the first post-cut system send from i to j, and let $send(j,i,m')$ be the first post-cut system send from j to i. Since there are no messages, either system or protocol, sent to j by i prior to m, Lemma 14 of Section 8.1.1 can be invoked to show that $e_i \rightarrow send((j,i,m'))$; the same kind of argument shows that $e_j \rightarrow send((i,j,m))$. Since there are no receive events between e_j and $send(j,i,m')$, or between e_i and $send(i,j,m)$, Lemma 5 implies $e_i \rightarrow e_j$ and $e_j \rightarrow e_i$. Thus we have reached a contradiction, and therefore either i must send a message to j in r or vice versa.∎

Recall that if $(i,j) \in \mathcal{C}$ then $(j,i) \in \mathcal{C}$; i.e. $|\mathcal{C}|$ is actually twice the number of channels in the system. The protocols $Flood1$ and $Flood2$ of Chapter 3 both then use $|\mathcal{C}| - (N-1)$. To see this, note that the initiator sends a message along each incident channel; every other processor sends a message along all but one incident channels. Thus there will be two messages on all but $N-1$ channels. These protocols can be modified to perform better in most cases. For example, in *Flood2*, let the initiator add the list $\{I\}$ to the Cut messages it sends out; let every processor p add itself to the list L which accompanies the first Cut message it receives and append this new list to all Cut messages it sends out. With this modification, a processor need send Cut messages and processor list only to those of its neighbours which do not appear in L. This modified protocol will require fewer messages in most instances; however, in any run in which the first Cut message received by each processor is the one sent by the initiator, there will still be $|\mathcal{C}|-(N-1)$ protocol messages sent. The quantity $|\mathcal{C}|-(N-1)$ does provide an exact lower bound on locally inhibitory CCPs which must work for all FIFO systems: in any system, $N-1$ messages are required simply to ensure that a protocol event occurs at each processor, and in a system in which the processor network is a tree $N-1=|\mathcal{C}|-(N-1)$. Though we strongly suspect that this quantity is indeed a precise lower bound for all classes of FIFO systems, we have as yet no proof of it.

8.3 Summary

We have analyzed the cost in message complexity of consistent-cut protocols for both non-FIFO and FIFO systems. We have proved an $O(N)$ lower bound on globally inhibitory CCPs in both cases. We have also illustrated a trade-off between the degree of inhibition used and the number of messages required for FIFO protocols: any locally inhibitory CCP must use $O(|\mathcal{C}|)$ messages.

Chapter 9

Conclusions and Related Work

In this work we have analyzed the cost of achieving causal consistency in asynchronous distributed systems. We have assessed this cost in terms of the degree of inhibition and the message complexity of consistent-cut protocols. Concerning inhibition, we have distinguished local versus global inhibition , and have consequently defined a spectrum of protocol capabilities with respect to inhibition. We have given a complete analysis of the existence of consistent-cut protocols as a function of these capabilities, while also considering the FIFO or non-FIFO nature of the system and whether or not protocol messages are allowed to be inconsistent with respect to a cut. We have shown that local inhibition is necessary and sufficient to develop a CCP for FIFO systems, while global inhibition of send events is necessary and sufficient for non-FIFO systems. Concerning message complexities, we have established that for non-FIFO systems, and CCP must use $O(N)$ messages. Any CCP for non-FIFO systems must use global inhibition; we show that globally inhibitory protocols for FIFO systems also have a $O(N)$ lower bound. Locally inhibitory CCPs, on the other hand, require $O(|\mathcal{C}|)$

messages, where $\mathcal{C}$ is the channel set of the system; this illustrates a trade-off between message complexity and the degree of inhibition used. Related work [Awe85] examines message versus time complexity trade-offs in a class of protocols, called *synchronizers*, which resemble CCPs although they differ somewhat in their causal constraints. A message-complexity lower bound for synchronizers is given as a decreasing function of time complexity. However, in that work inhibition is not considered directly, although it is a potential source for increasing the time complexity of protocols.

Another related issue is that of piggybacking. Non-FIFO channels in which messages can be "piggybacked" differ from the pure non-FIFO case, because an order is imposed on messages that are packaged together. The "red-white" algorithm of [LY87,Mat89] is an example of a protocol which piggybacks a protocol marker onto system messages in order to determine consistent cuts. In that protocol, a color—either white or red—is associated with each processor at each point in its local history. A processor starts out white, but may spontaneously turn red, after which it adds a red marker to each message it sends to a neighbor. Any white processor about to receive a red marker turns red immediately before doing so. The white states of the processors form a consistent cut.

Our definition of "inhibitory" assumes a model in which system and protocol events are distinct, therefore it is difficult to apply it to the red-white protocol. However, since the protocol only adds to, and does not impede, the sending or reception of any system message, it could reasonably be considered "non-inhibitory." Thus one could take the viewpoint that the red-white protocol is a non-inhibitory, consistent-cut protocol for non-FIFO systems; although, as we will observe below, it does not quite satisfy our conditions on CCPs. Regardless, we have shown in Theorem 1 that without allowing piggybacking or some other mechanism stronger than those in our model, such a protocol is not possible to attain.

The red-white protocol, while certainly determining consistent global states, does not satisfy the "distinguishability" requirement (condition (2)) of our definition of a CCP. A particular local state may be the cut state of processor p in one run (because the next event at p in that run is the reception of a red message) but not in another (because the following event is, say, some internal event enabled by that local state). In effect, the protocol designates a *previous* state as a processor's cut state; the processor does not know until after the fact that it was at the cut. Alternatively, one could assume that a processor is able to "peek" at the contents of arriving messages. Then a processor could, immediately before the reception of a red message, designate its current state as the cut state. This requires stronger capabilities for processors than those granted in our model.

Another approach is to model the reception of a piggybacked message in the red-white protocol as two separate but consecutive receive events, the first of the marker and the second of the system message (note that this is FIFO behavior). The red-white algorithm could then be modified to designate the state ending with the reception of the red marker as a processor's cut state. This protocol satisfies condition (2) of our CCP definition, and has inconsistent protocol messages. However, if we do model the reception of piggybacked messages as distinct receive events, guaranteeing their consecutive reception would seem inherently to require some form of inhibition. This becomes more acute if one looks at a final approach to designing CCPs using piggybacking: that of simulating FIFO channels by including the entire message history with every message. Note that this also requires unbounded message size. In sum, the modeling of piggybacked messages, and their interaction with distinguishability and inhibition, involve many subtle issues.

Message complexity issues are also affected if piggybacking is introduced. There need be no distinct protocol messages of course; instead, we might count the number of protocol markers used. However, any theorems which invoke arguments on the lines of "there must be a protocol message to prevent the inconsistency of a system event which *might* occur later" are rendered invalid: a protocol need not guard against *possible* system sends, as it has only to piggyback protocol markers on *actual* system messages. The question of what is meant by the termination of such a protocol must then also be addressed.

Bibliography

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the A.C.M.*, 32(4), October 1985.

[BF88] L. Bougé and N. Francez. A compositional approach to superimposition. In *Proceedings of the A.C.M. Symposium on Principles of Programming Languages*, January 1988.

[BJ87] Ken Birman and Thomas Joseph. Reliable communication in the presence of failures. *A.C.M. Transactions on Computer Systems*, 5(1), February 1987.

[Bra80] W. Brauer(ed.). *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[Cha82] E.J.H. Chang. Echo algorithms: Depth parallel operations on graphs. *I.E.E.E. Transactions on Software Engineering*, SE-8(4):391–400, 1982.

[CL85] M. Chandy and L. Lamport. Finding global states of a distributed system. *A.C.M. Transactions on Computer Systems*, 3(1):63–75, 1985.

[CM85] M. Chandy and J. Misra. How processes learn. In *Proceedings of the Fifth A.C.M. Symposium on Principles of Distributed Computing*, pages 204–214, 1985.

[Cri89] Carol Critchlow. On inhibition and atomicity in asynchronous consistent-cut protocols. Technical Report 89-1069, Cornell University Department of Computer Science, December 1989.

[CT90a] C. Critchlow and K. Taylor. The inhibition spectrum and the achievement of causal consistency (extended abstract). In *Proceedings of the Ninth A.C.M. Symposium on Principles of Distributed Computing*, pages 31–42, 1990.

[CT90b] C. Critchlow and K. Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report 90-1101, Cornell University Department of Computer Science, February 1990.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the A.C.M.*, 34(1):77–97, January 1987.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* Ph.D. dissertation, Carnegie Mellon University, 1988.

[Fel79] J.A. Feldman. High level programming for distributed computing. *Communications of the A.C.M.*, 22(6):353–368, June 1979.

[Fra80] N. Francez. Distributed termination. *A.C.M. Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[Gel85] David Gelernter. Generative communication in Linda. *A.C.M. Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the A.C.M.*, 37(3):549–587, July 1990.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the A.C.M.*, 21(8):666–677, February 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, London, 1985.

[Jos91] Mark B. Josephs. Receptive process theory. Technical Report (to appear), Department of Mathematics and Computing Science, Eindhoven University of Technology, 1991.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *I.E.E.E. Transactions On Software Engineering*, SE-13(1):23–31, January 1987.

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

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, M.I.T. Laboratory for Computer Science, April 1987.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *C.W.I. Quarterly*, 2(3):217–246, September 1989.

[LY87] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, 1987.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms* (Proceedings of the International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, October 1988), M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, Editors, pages 215–226. Elsivier Science Publishers (North-Holland), 1989.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Pet66] C.A. Petri. *Kommunikation mit Automaten*. Ph.D. dissertation, Institut für Instrumentelle Mathematik, BONN, 1966.

[PT88] P. Panangaden and K. Taylor. Concurrent common knowledge: A new definition of agreement for asynchronous systems. In *Proceedings of the Seveth A.C.M. Symposium on Principles of Distributed Computing*, pages 197–209, 1988.

[PT89] P. Panangaden and K. Taylor. Concurrent common knowledge: A new definition of agreement for asynchronous systems. Technical Report 89-1011, Cornell University Department of Computer Science, May 1989.

[Rus77] D. L. Russell. Process backup in producer-consumer systems. In *Proceedings of the A.C.M. Symposium on Operating Systems Principles*, November 1977.

[Tay89] Kim Taylor. The role of inhibition in asynchronous consistent-cut protocols. In *Lecture Notes in Computer Science 392: Distributed Algorithms* (Proceedings of the Third International Workshop on Distributed Algorithms, Nice, France, September 1989), J.-C. Bermond and M. Raynal, Editors, pages 280–291. Springer-Verlag, 1989.

[Win86] Glynn Winskel. Event structures. Technical Report 95, University of Cambridge, Computer Laboratory, 1986.