Knowledge and Inhibition in Asynchronous Distributed Systems

Kimberly E. Taylor
Ph.D Thesis

TR 90-1139
July 1990

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

KNOWLEDGE AND INHIBITION IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Kimberly Elaine Taylor
August 1990

KNOWLEDGE AND INHIBITION
IN ASYNCHRONOUS
DISTRIBUTED SYSTEMS

Kimberly Elaine Taylor, Ph.D.
Cornell University 1990

In an asynchronous distributed system, processes communicate only via message passing along channels with unbounded transmission time. Relative process speeds are arbitrary and processes do not have access to a common clock. For such systems, a useful means of providing temporal structure is the *causality* relation. Causality imposes a partial order on the events of an execution based on the necessity of certain events preceding other events. Causality can be used to give a sensible definition of a distributed global state in an asynchronous system, known as a *consistent cut.*

We present a new knowledge-theoretic logic, with formal semantic definitions, for reasoning about causality and consistent cuts. The logic includes *concurrent common knowledge*, a new form of distributed agreement, which has an analogous role to that of common knowledge in synchronous systems. It is shown to be a necessary and sufficient condition for performing concurrent actions in asynchronous systems, and has a role in many distributed applications such as checkpointing, deadlock detection, and broadcasting. Concurrent common knowledge is also shown to be attainable by class of protocols termed *distinguishable-consistent-cut protocols*, or *DCCPs.*

We examine four simple and efficient DCCPs and trade-offs between them. These trade-offs involve message complexities, necessity of FIFO channels, and the degrees to which events of the underlying system are suspended during execution of the protocols. We call such suspensions *inhibition* and formally define multiple forms of inhibition that are used in distributed protocols. We classify the DCCPs in this work according to their inhibitory characteristics, and prove new results which demonstrate close relationships between inhibition and the existence of DCCPs.

Biographical Sketch

Kimberly Elaine Taylor was born amidst the rolling green hills and country-music wax museums of Nashville, Tennessee, at Baptist Hospital on June 3, 1963. In 1976 she moved to the small and friendly town of Mt. Juliet, Tennessee, world-renowned at that time in history for Charlie Daniels' Day. Kim graduated as Valedictorian of Mt. Juliet High School in May of 1981.

Despite several glamorous years as a waitress at Cedar Creek Country Club and occasional dreams of being a pianist, she decided to go to college and, at random, to study computer science. Anxious to see the "real world," she instead ended up in Texas. In spite of Houston, she spent four happy years at Rice University, where her claim to fame was that she could chug twelve ounces of warm, flat beer in under four seconds as part of the utterly dominant Will Rice College women's Beer-Bike team. Kim graduated Cum Laude from Rice in May of 1985, earning the B.S.E.E. in electrical engineering and computer science.

Despite warnings from all of her friends, who were bound for northern California and high-paying jobs, after college Kim decided to attend graduate school at Cornell University in Ithaca, New York. She did, however, lead a pleasant graduate student life and earned many, many frequent-flyer awards during that period; not to mention being pianist for the infamous Steamin' Weenies. Kim earned the M.S. in computer science from Cornell in May 1988, and the Ph.D. in computer science in August 1990. In September of 1990 she will join the faculty of the University of California at Santa Cruz as Assistant Professor in the Division of Computer and Information Sciences. She looks forward to living on sunshine, sushi, salt water, and NSF grants.

To my mother, Estelle Crismon Moon, for her love, support, and encouragement all these years.

Acknowledgements

I first and foremost wish to thank Prakash Panangaden for being an unsurpassable advisor, mentor, and friend throughout the past five years. In particular, for listening to and even encouraging the most nebulous of ideas, for having faith in me long before I had faith in myself, and for allowing me the freedom to develop as a whole person in addition to as a researcher.

I wish to thank the entire distributed systems community at Cornell, especially Sam Toueg and Ken Birman, for their support and encouragement. I also appreciate having had the pleasure of working with Carol Critchlow in our latest research.

I thank John Wilkes at Hewlett-Packard Laboratories for his support and enthusiasm and for giving me the opportunity to experience computer science outside of the ivory towers. I am also grateful to Cornell University and to AT&T Bell Laboratories for their fellowship support.

Last but definitely not least, I cannot possibly thank enough the many friends—in particular Bruce Hendrickson, Kathy Everette, Andrew Westphal, Michael Slifker, and Mike Quayle—who have been extremely supportive and made graduate life infinitely more pleasant by their presence.

Table of Contents

List of Figures

Chapter 1

Introduction

This work is an investigation into the structure and capabilities of asynchronous, distributed systems. In an asynchronous distributed system, processes communicate only via message passing along channels with unbounded transmission time. Relative process speeds are arbitrary and processes do not have access to a common clock.

When distributed processes cooperate to accomplish tasks, it is typically necessary to coordinate when operations on one process occur relative to operations on other processes. In an asynchronous system, physical time cannot be used for such coordination. Instead, a useful means of providing temporal structure is the *causality* relation [Lam77]. Causality imposes a partial order on the events of an execution based on the necessity of certain events preceding other events. Causality can also be used to give a sensible definition of a distributed global state in an asynchronous system, known as a *consistent cut.* A consistent cut is reasonable in this context because, in an asynchronous system, processes cannot distinguish that such a state did or did not occur instantaneously.

In recent years, knowledge-theoretic logic [HM84] has become a popular tool for reasoning about and proving properties of general distributed systems. The standard logic of knowledge for distributed systems includes a definition of *common knowledge*, a state of "agreement" between processes. However, common knowledge is attainable in a system if and only if simultaneous action is possible. Consequently, this form is unattainable asynchronously, and therefore not useful for describing coordination of distributed tasks in an asynchronous system.

We present a new knowledge-theoretic logic, with formal semantic definitions, designed specifically for reasoning about asynchronous systems. Our semantic definitions are consequently based on causality and consistent cuts. The logic includes *concurrent common knowledge*, a new form of distributed agreement, which has an analogous role to that of common knowledge in synchronous systems. We will show that it is a necessary and sufficient condition for performing concurrent actions in asynchronous systems, and has a role in many distributed applications such as checkpointing, deadlock detection, and broadcasting. Concurrent common knowledge is shown to be attainable by a class of protocols termed *distinguishable-consistent-cut protocols*, or *DCCPs*.

We examine four simple and efficient DCCPs and trade-offs between them. These trade-offs involve necessity of FIFO channels, message complexities, and the degrees to which events of the underlying system are suspended during execution of the protocols. We formally define *inhibition* to describe the behavior by a protocol of suspending underlying system events. We further distinguish *local* versus *global* inhibition; global inhibition requires communication from other processors to "wake up" suspended events whereas local inhibition does not. We classify the DCCPs in this work according to their inhibitory characteristics, and demonstrate close relationships between inhibition and the existence of DCCPs. In particular, we show that the global inhibition of send events is necessary for developing a DCCP for non-FIFO systems.

1.1 Previous Work

We first discuss two areas of distributed systems research upon which our work builds. The first is causality, which is used to give temporal structure to asynchronous distributed systems. The second is knowledge-theoretic logic, which is used to reason about communication and cooperation in distributed systems.

We also examine other weakenings of common knowledge that are based on replacing simultaneity with an alternative temporal notion. Two of these are attainable asynchronously, and are therefore of particular relevance to our work.

1.1.1 Causality

As noted previously, it is not sensible to use physical time to relate when events occur relative to each other in asynchronous systems. Instead, Lamport [Lam77] suggested that the only temporal relationships observable in such a system are those imposed by sequential execution on single processes and those imposed by communication, i.e. the sending and receiving of messages. The *happens-before* relation was defined to describe those temporal relationships, as follows.

We assume that we have a set of *events* to describe the operations performed by each process, which includes the sending and receiving of messages; we will give a more formal description in Chapter 2. Happens-before is a relation containing pairs of events. Given two events e_1 and e_2 in an execution of a system, e_1 happens-before (denoted $\rightarrow$) e_2 if any of the following are true:

1. Events e_1 and e_2 are operations performed by a single processor, and e_1 is performed earlier than e_2.

2. Event e_1 is the sending of a message and e_2 is the reception of the same message.

3. The pair (e_1, e_2) is in the transitive closure of the set of pairs satisfying conditions (1) and (2) above; in other words, there is a sequence of pairs of events $(e_1, e_{x1}), (e_{x1}, e_{x2}), \ldots, (e_{xk}, e_2)$ such that either condition (1) or condition (2) above holds for each pair.

The fact that event e_1 happens-before e_2 reflects *potential causality*, that is, the execution of event e_1 could (but does not necessarily) have a causal effect on the execution of event e_2.

The temporal notion of causality was also used by Lamport to give a sensible definition of distributed *global state* in an asynchronous system. A global state is a collection of local states, one per process. In a synchronous system, an *instantaneous* global state is typically used. An instantaneous global state is a set of local states that all occur at the same physical time. Again, since the processes in an asynchronous system cannot coordinate simultaneously, such a definition has limited usefulness. Instead, Lamport proposed *consistent cuts* as meaningful global states in distributed asynchronous systems.

A consistent cut is defined so that, if event e_1 happens-before event e_2 and e_2 is in the cut, then e_1 is also in the cut. In other words, an event cannot have occurred in a consistent cut unless events which could have causally affected it have occurred also. This results in a global state which *could* have happened simultaneously; that it did or did not is not observable by processes due to the asynchrony in the system. In the case of multiple consistent cuts that intersect—some events are in one but not the other, and vice versa—it may be observable that *all* such cuts did not occur simultaneously; however, it cannot be determined that any one of those cuts was not an instantaneous global state.

Consistent cuts are used frequently in protocols for asynchronous distributed systems. Some examples are in checkpointing [CL85,KT87], deadlock detection [BT87], distributed termination [Fra80], and broadcasting [BJ87].

Another concept related to causality, also introduced by Lamport [Lam77], is that of *logical clocks.* A clock in this context is a function that assigns a value to each event of an execution. We denote the value associated with event e by $clock(e)$. We assume further that local states are also associated with clock values, and that the clock values of events and the states in which they occur—more accurately, in our model, the states that they follow—agree. Logical clocks have the property that they do not conflict with causality; i.e. if $e_1 \rightarrow e_2$ then $clock(e_1) < clock(e_2)$. Consequently, if a set of local states have the same logical clock value, then they form a consistent cut. The converse of the past two statements do not hold: if $clock(e_1) < clock(e_2)$ then it may not be the case that $e_1 \rightarrow e_2$, and a consistent cut may contain states having different logical clock values. Logical clocks are relevant to our work when comparing concurrent common knowledge to timestamped common knowledge, in the case where timestamps are logical clock values as in [NT87].

1.1.2 Knowledge

Logics of knowledge have recently been proposed [HM84] as a tool for reasoning about communication and cooperation in distributed systems. The motivation for this is a natural tendency to reason informally by describing what processes "know" about other processes' states based on their local information. Knowledge theory gives a way of formalizing this notion. It is important because, while coordinating distributed tasks involves the states of all processes, the performance of an individual action is based solely on the local state of a single process.

Knowledge is formalized by giving a set of "possible worlds" and an indistinguishability relation on those worlds for each agent involved [Leh84]. In Halpern and Moses, the set of possible worlds is the set of instantaneous global states which may occur in executions of a system. The agents, of course, are processes. Two global states are indistinguishable to a process if and only if the process has the same local state in each. A process is said to *know* a fact in a particular global state if the fact is true in all global states that are indistinguishable from the current one to that process. The following notation is used:

$$(r,t) \models K_i\phi$$

In the formula above, (r,t) denotes an instantaneous global state of an execution; r for "run" indicates the execution, and t denotes the physical time. K_i represents "i knows," and ϕ denotes some fact.

Next, forms of "group knowledge" are defined; two are of particular interest: *everyone knows*, denoted E, and *common knowledge*, denote C. As expected, everyone knows ϕ is true in a global state iff all processes know ϕ in that state. Below, $\mathcal{I}$ denotes the set of all processes.

$$E\phi \equiv \bigwedge_{i\in\mathcal{I}} K_i\phi$$

It would seem, perhaps, that "everyone knows" of relevant facts would be sufficient for agreement in a system. This is not true, however. The following "generals paradox" is a variant from classic operating systems theory [YC79, Gra78].

> Suppose that there are three generals with three armies, one in a valley and the others on two hills overlooking the valley. The two armies in the hills wish to defeat the army in the valley. It is known that if they attack at the same time then they will win. However, it is also known that if either attacks alone then that army will be

> defeated. The two armies can only communicate by a messenger on foot. The messenger is reliable in that she will eventually deliver any message; however, she may take arbitrarily long to make a delivery.
>
> It appears that the two armies can coordinate through the use of their messenger and by using the sun as a global reference point. So, one general sends the messenger to the other general, instructing the armies to attack at dawn the next day. Recall that the messenger may take arbitrarily long to make a delivery. However, on this occasion she arrives in a timely manner, well before dawn. Will the armies attack?

Neither army will attack in the situation above, although upon reception of the message they both know that they are supposed to attack at dawn. The problem is that the sender of the message does not know that the message was delivered in time; in other words, the first army doesn't know that the second army knows to attack at dawn. Even if the messenger returns an acknowledgement, and she arrives well before dawn, there is still uncertainly with respect to whether or not the acknowledgement actually arrived.

It turns out that simultaneous action of this type requires that everyone knows a fact, and everyone knows that everyone knows it, and everyone knows that everyone knows that everyone knows it, and so on ad infinitum. Such a knowledge state is called common knowledge.

$$C\phi \Leftrightarrow E\phi \wedge E^2\phi \wedge E^3\phi \wedge E^4\phi \ldots$$

The actual semantic definition of common knowledge relates logical formulas to sets of global states which satisfy them.

Common knowledge has proven to be useful for reasoning about communication in synchronous systems [DM86,MT88]. However, common knowledge is unattainable in asynchronous systems [HM84] because one can show that it must be attained simultaneously by all processes. This motivated our development of an alternative definition of agreement for asynchronous systems.

1.1.3 Other Knowledge Forms

In addition to concurrent common knowledge, there have been other knowledge-theoretic definitions for agreement that weaken the need for simultaneity. Two of these are also achievable asynchronously: eventual common knowledge and timestamped common knowledge [HM84]. In Section 4.4 we give a detailed comparison of these definitions to concurrent common knowledge.

Eventual common knowledge is incomparable to concurrent common knowledge in a strict sense: there are systems in which each is attainable while the other is not. The same is true, in general, for timestamped common knowledge. However, timestamped common knowledge in systems that use logical clocks [Lam77, NT87] and in which the timestamp of interest is guaranteed to be reached by all processes is strictly stronger than concurrent common knowledge. Neiger and Toueg give a characterization of problems for which timestamped common knowledge is a suitable replacement for common knowledge in asynchronous systems. Timestamped common knowledge implies a useful property for some distributed applications which concurrent common knowledge does not; namely, if two facts become timestamped common knowledge independently then all processors attain the associated knowledge for each fact in the same order. However, this ordering property is not necessary in many distributed applications, and concurrent common knowledge can be attained with less latency than timestamped common knowledge. Furthermore, the definition and achievement of concurrent common knowledge is independent of any type of local process clock. Thus, each definition is sometimes the more appropriate knowledge form for a given application. We discuss this further in Section 4.4.5 and in Section 6.3.

1.2 Outline

Our presentation is organized as follows. The next chapter gives a formal description of our system model, to be used throughout the remainder of the work. In Chapter 3 we define distinguishable-consistent-cut protocols (DCCPs), and give four examples. DCCPs are also relevant to all subsequent chapters, as they have central roles in the discussions of both knowledge and inhibition. In Chapter 4, we give a logic with semantic definitions for concurrent common knowledge. We give a general condition for its attainability, and show that DCCPs can attain concurrent common knowledge of certain facts. We also compare this form of knowledge to other variants of common knowledge found in the literature. In Chapter 5, we define the notion of inhibition, and examine its relationship to the existence of DCCPs. In Chapter 6 we discuss some applications of concurrent common knowledge and DCCPs, and in Chapter 7 we give our conclusions.

Chapter 2

System Model

In this chapter we describe a model for the type of asynchronous, distributed system [Lam77] that is the focus of our work; thus it is a foundation for all subsequent chapters. We use the model to describe the behavior of distinguishable-consistent-cut protocols (Chapter 3), to give semantic definitions for concurrent common knowledge and prove its attainability (Chapter 4), and to define the notion of inhibition and its relationship to distinguishable-consistent-cut protocols (Chapter 5). It is the latter role in which many of the finer details of this model are particularly useful.

By the term *distributed*, we mean that the system is composed of a set of autonomous processes that communicate only by sending messages along a fixed set of channels. The network is connected but not necessarily completely connected. By *asynchronous*, we mean that: (1) there is no global clock in the system, (2) there are no bounds on message transmission time, and (3) there are no assumptions made concerning the relative speed of processors. We will also assume that processors do not fail and that the communication medium never introduces spurious messages or alters the contents of messages. For the most part we will consider *reliable* channels, in which messages are not lost.

It is our intention to give a definition of the model that uses only the structures that are relevant to asynchronous systems. Thus we do not use timed runs to describe these systems as do Halpern and Moses [HM84] or Halpern and Fagin [HF88]. The resulting definitions turn out to be more natural than if we had detailed timing information in the model description. Of course, the timed runs model is more generally applicable than ours, precisely because it contains the additional timing information. In particular, it is possible to define $C^{\diamond}$, eventual common knowledge, and C^{ϵ}, epsilon common knowledge, using the timed-runs model. In Section 4.3 we give a precise translation between our formalism and the timed runs formalism.

The model is very similar to that of Chandy and Misra [CM86]. A system is characterized by a set of possible executions, or *runs*. Runs are composed of sequences of *events* corresponding to the local operations of individual processes. Whereas [CM86] uses a totally-ordered event sequence to represent a run, we use partially-ordered sequences. A partially-ordered run corresponds to many possible totally-ordered runs and, for asynchronous systems, captures the observable temporal relations between events.

Unlike [CM86], we explicitly include the *enabling* of events by past event sequences [Win86]. We also explicitly separate protocol events from underlying system events. These two features allow us, in Chapter 5, to define *non-inhibitory* protocols as those whose events never disable those of the underlying system.

2.1 Preliminary Definitions

The description of a distributed system is based on the behaviors of the individual components or *processes* in the system. A finite sequence of natural numbers is used to identify processes.

Definition 1. The *process identifier set*, $\mathcal{I}$, is a sequence of natural numbers $1, 2, \ldots, N$, for some finite N.

We take the notion of a *local state* of a process to be primitive. We let, for each $i \in \mathcal{I}$, $States_i$ denote the set of all possible local states of process i. Each process is also assumed to have a distinguished *initial state*, denoted s_i^0.

Actions, as in [Lam85,HF88], are state transformers.

Definition 2. An *action* is a function from local states to local states. There are three types of actions: *send actions* denoted *send*(m) where m is a message (described later), *receive actions* denoted *receive*(m), and *internal actions.*

Associated with each process i is the set Act_i of actions, corresponding to operations that can be performed by that process.

We use local states and actions to compose *local histories.*

Definition 3. A *local history*, l_i, of process i, is a (possibly infinite) sequence of alternating local states—beginning with the initial state—and elements of Act_i. We write such a sequence as follows.

$$l_i = s_i^0 \xrightarrow{\alpha_i^1} s_i^1 \xrightarrow{\alpha_i^2} s_i^2 \xrightarrow{\alpha_i^3} s_i^3 \ldots$$

We use s_i^j (α_i^j) to refer to the j^{th} state (action) in process i's local history.

An instantiation of an action, i.e. a specific state transition of that action, is termed an *event.*

Definition 4. An *event* is a state transition $\langle s_i^{j-1}, \alpha_i^j, s_i^j \rangle$ and is denoted e_i^j.

The set $Events_i$ contains all possible events corresponding to actions in Act_i.

We denote *receive events* and *send events* as for actions, with an additional parameter to uniquely identify instantiations: $send(k, m)$ and $receive(k, m)$, where k is a natural number and m is a message. We will require that the k^{th} occurrence of the sending of a particular message in a local history have instantiation number k; a similar requirement for receives will depend upon the channel behavior. The instantiation number is only part of the model and is used to describe certain behaviors in the remainder of this section; however, it is not necessarily a part of the messages themselves. Instantiation numbers in messages would require that they be arbitrarily long; we do not impose such a requirement.

The state of a process can be obtained from its initial state and the sequence of actions or events that have occurred up to the current state. Hence the local history may be equivalently described as either of the following:

$$l_i = s_i^0, \alpha_i^1, \alpha_i^2, \alpha_i^3 \ldots$$

$$l_i = s_i^0, e_i^1, e_i^2, e_i^3 \ldots$$

where $e_i^j = \langle s_i^{j-1}, \alpha_i^j, s_i^j \rangle$. If it is additionally assumed that the local state includes a description of all past actions in the local history (corresponding to the *complete history* interpretation of [HM84,HF88]), then

$$l_i = s_i^0, s_i^1, s_i^2, s_i^3 \ldots$$

is also equivalent to the above formulas. We will assume such an interpretation; note that this interpretation results in a maximum amount of information being available to a process based on its local state.

We often omit the subscript or superscript on states, events, and actions when it is obvious or irrelevant. Concatenation of events to produce new local states is denoted by "$\cdot$" as in $s = s' \cdot e$. If local state s is a prefix of local history l, we say that predicate $Prefix(s, l)$ is true.

Finally, we define *messages* and *channel sets.*

Definition 5. A *message* is a triple $\langle i, j, B \rangle$ where $i \in \mathcal{I}$ is the sender of the message, $j \in \mathcal{I}$ is the message recipient, and B is the body of the message. B can be either a special value (e.g. a tag to denote a special-purpose message), or some proposition about the run (e.g. "i has reset variable X to zero"), or both.

Definition 6. A *channel set* is any set of ordered pairs of elements of $\mathcal{I}$, i.e. any subset of $\{(i, j) : i, j \in \mathcal{I}\}$. Channels are uni-directional; the occurrence of (i, j) in a channel set indicates that process i can send messages to process j but not necessarily vice-versa.

2.2 Enabling

An *enabling relation* for each process is used to describe when it is possible for an action or event to be executed by the process as a function of its local state. Consequently, it contains pairs consisting of local states and events. (Note that, since the local state is specified, describing the enabling of either an action or an event is equivalent. We have chosen to use events.)

Definition 7. En_i is an *enabling relation* for process i iff

$$En_i \subseteq States_i \times Events_i.$$

If $(s_i, e_i) \in M_i$ then we say s_i *enables* e_i. The enabling relation describes what events *could* occur in a given local state of process i; there may be several events enabled by a single local state.

We require the following: for each process i and each local state s_i of i, if $(s_i, receive(m)) \in M_i$, where m is a message from process j, then $(s_i, receive(m')) \in M_i$ for any message m' sent by j. This means that at any point, i may be willing to receive messages at one channel and not another, but may not selectively enable receives on a single channel based on message contents.

2.3 Causality

In an asynchronous system, physical-time differences between occurrences of events on different processes are not distinguishable by processes and cannot be used to coordinate actions. There is, however, a distinguishable ordering of the events of a system execution due to the fact that certain events are known to precede—thus potentially affecting—other events. We can define this order using *potential causality* as done by Lamport [Lam77]. In an asynchronous system, potential causality results only from sequential execution on single processes and from message passing between separate processes. It is described using the *happens-immediately-before* relation $\mapsto$ and the *happens-before* relation $\rightarrow$.

Definition 8. Given an N-vector of local histories, event e_i^x *happens-immediately-before* event e_j^y, denoted $e_i^x \mapsto e_j^y$, if and only if (1) e_i^x and e_j^y are different events in the history of some process i and e_i^x occurs first in the sequence, i.e. $i = j$ and $x < y$, or (2) e_i^x is an instantiation of the sending of a message and e_j^y is the same instantiation of the reception of that message; i.e. there exists m and k such that $e_i^x = send(k, m)$ and $e_j^y = receive(k, m)$.

Definition 9. The *happens-before* relation, denoted $\rightarrow$, is the transitive closure of happens-immediately-before.

Thus if $e_i^x \rightarrow e_j^y$, then either $e_i^x \mapsto e_j^y$ or there exists an event e_k^z such that $e_i^x \rightarrow e_k^z$ and $e_k^z \mapsto e_j^y$.

2.4 System Model

We can now define our system model as follows.

Definition 10. A *system* $\mathcal{S}$ is a quadruple $\mathcal{S} = (\mathcal{I}, \mathcal{C}, Act, En)$ consisting of a process identifier set $\mathcal{I} = \{1, ..., N\}$, a channel set $\mathcal{C}$, an N-vector of action sets $Act = \{Act_1, Act_2, \ldots, Act_N\}$, and an N-vector of enabling relations $En = \{En_1, En_2, \ldots, En_N\}$ such that (1) for each process i, En_i is an enabling relation for i, (2) for each message from process i to process j in Act, (i, j) is in $\mathcal{C}$, and (3) for any two processes i and j, there is a path $(i, i_1), (i_1, i_2,), ..., (i_k, j)$ in $\mathcal{C}$ (the network is connected).

From a system $\mathcal{S}$ we can construct a set of possible *asynchronous runs*, denoted $\mathcal{A}(\mathcal{S})$, of the system. First of all, we define the notions of *partial runs* and *total runs*.

Definition 11. Given a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, Act, En)$, a *partial run of* $\mathcal{S}$ is an N-vector of local states, $\langle s_1, s_2, \ldots, s_N\rangle$, such that the following conditions hold:

1. For every receive there is a corresponding send, i.e. if $receive(k, \langle i, j, \phi\rangle)$ is in s_j then $send(k, \langle i, j, \phi\rangle)$ is in s_i.
2. For no two events e_1 and e_2 does $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_1$.
3. For all events there is an enabling state (possibly the initial state) immediately before it in the local state, i.e. if $e_i^j = \langle s_i^{j-1}, \alpha_i^j, s_i^j\rangle$ then $(s_i^{j-1}, e_i^j) \in En_i$.

Total runs are sets of local histories which satisfy the conditions on partial runs above and, in addition, are complete in the sense that no process ceases execution while there are enabled send or internal events or enabled receive events for which there is a corresponding send event in the run. We will often refer to a total run as simply a "run."

Definition 12. Given a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, Act, En)$, a *total run of* $\mathcal{S}$ is an N-vector of local histories that is a partial run and in which, for every local history l_i, either (1) l_i is infinite, or (2) if $(l_i, e_i) \in M_i$ then e_i is the reception of a message from process j and there is no corresponding send event in l_j.

Finally, $\mathcal{A}(\mathcal{S})$, the *asynchronous runs of* $\mathcal{S}$, is the set of all total runs of $\mathcal{S}$. We typically use a to denote an asynchronous run in the set $\mathcal{A}(\mathcal{S})$. We let $a[i]$ denote the local history of i in run a. We say that run a *extends* partial run a' iff $\forall i \,.\, Prefix(a'[i], a[i])$.

Our requirements on asynchronous runs are equivalent to those in [CM86], with the exceptions of the enabling constraints and that we limit message sending to occur along the set of designated channels. Chandy and Misra express the possible behaviors of systems in terms of totally ordered sets of events called *system computations.* Their conditions on system computations are that (i) projections on each process are possible local histories, and (ii) the reception of a message is preceded by its sending. These are equivalent to stating that the system computations are linearizations of the $\rightarrow$ relation.

We introduce three optional constraints on the runs of a system: reliability, FIFO, and duplex. Reliability says that if a message is sent then it is received, i.e. no message loss occurs. FIFO indicates that channels exhibit first-in-first-out behavior. Duplex indicates that, if i can send a message directly to j, then j can send a message directly to i. These properties are not necessary for our definitions, but we will want to address systems that satisfy them when we address the attainability of concurrent common knowledge. Unless otherwise stated, they will not be assumed in the model.

Reliability: $send(k, \langle i, j, \phi \rangle) \in l_i \Rightarrow receive(k, \langle i, j, \phi \rangle) \in l_j$

FIFO: If $\alpha_i^w = send(k_1, \langle i, j, \phi_1 \rangle)$, $\alpha_i^x = send(k_2, \langle i, j, \phi_2 \rangle)$, and $w < x$, then there exists actions $\alpha_j^y = receive(k_1, \langle i, j, \phi_1 \rangle)$ and $\alpha_j^z = receive(k_2, \langle i, j, \phi_2 \rangle)$ such that $y < z$.

Duplex: If $(i, j) \in \mathcal{C}$ then $(j, i) \in \mathcal{C}$.

2.5 Consistent Cuts

Causality can be used further to describe global states of an asynchronous system. A global state is some prefix of a run, as defined below.

Definition 13. A *global state* of run a is an N-vector of local states of a, one per process.

A *consistent* global state [Rus77,CL85], sometimes termed a *consistent cut*, is described by causality as follows. Equivalently, in our model, a global state is consistent if it is a partial run.

Definition 14. A *consistent cut* of a system is any global state such that if $e_i^x \rightarrow e_j^y$ and e_j^y is in the global state, then e_i^x is also in the global state.

(See Figure 2.1. The states of (b) form a consistent cut whereas those of (a) do not.) We will denote a consistent cut by a vector of local states $c = \langle s_1, \ldots, s_N \rangle$; we also use the notation $c[i]$ to indicate the local state of i in cut c.

For simplicity throughout the remainder of this work, we will not explicitly mention the instantiation number of events.

Several of our proofs, including that of the following lemma, refer to causally-related *message chains* as defined below.

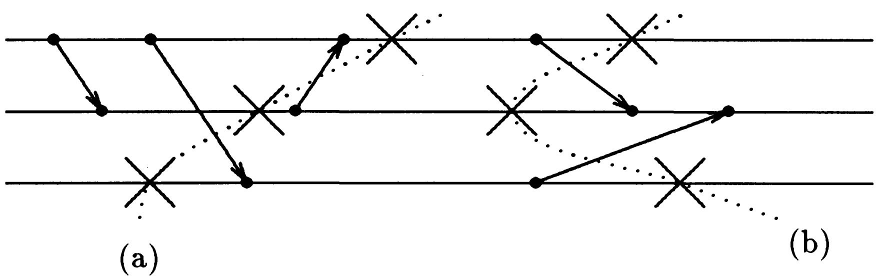

Figure 2.1: Inconsistent (a) vs. consistent (b) cuts.

Definition 15. In an asynchronous run, a *message chain* is a (possibly infinite) sequence of messages $m_1, m_2, m_3, \ldots$ such that, for all i, $receive(m_i) \rightarrow send(m_{i+1})$. Consequently,

$$send(m_1) \rightarrow receive(m_1) \rightarrow send(m_2) \rightarrow receive(m_2) \rightarrow send(m_3) \ldots.$$

Finally, the following lemma establishes a desirable property of consistent cuts in asynchronous runs.

Lemma 1. In any asynchronous run of any system, each local state of each process is included in some consistent cut of the system.

Proof: Fix an asynchronous run, a, of a system. Consider any local state of any process, say state s_x^i on process i. Let e_x^i be the last event executed in s_x^i. We will iteratively construct a consistent cut c of a such that $(a, c)[i]$ is equal to s_x^i. Let $MinSends(k)$ for any k be the minimum (earliest) local state of process k in run a which includes the sending of all messages received by i from k in local state s_x^i. Initialize a vector of local states c as follows: $c[i] = s_x^i$ and for any $j \neq i$, $c[j] = MinSends(j))$. We refer to this as the *initial state vector.* Of course this vector is not necessarily a consistent cut. On each step of the iteration, find a message m' from any process j to any other process k whose reception is in c but whose sending is not (if such a message doesn't exist then we are finished). Set $c[j]$ to be the minimal (earliest) local state in $a[j]$ which includes the sending of m'; note that the old $c[j]$ is a strict prefix of the original. In order to meet the conditions of the theorem, we must show that the iterations terminate and that the value of $c[i]$ never changes.

We first make an observation to be used extensively in the remainder of the proof: that, for any inconsistent message m' in the above construction, there is a message chain beginning with m' ending with a message received within the initial state vector. Consider any message m' as above, from process j to process k. Then either $c[k]$ is a member of the initial state vector or the last event of $c[k]$ is the sending of a message m'', where m'' was the inconsistent message of some

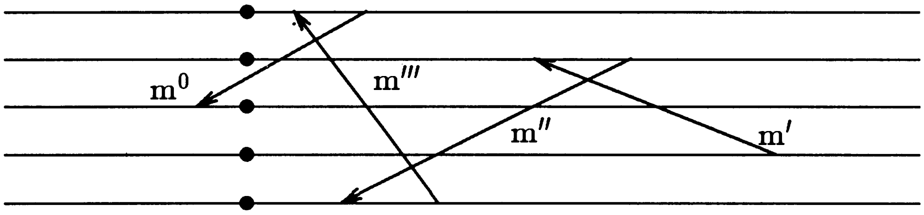

Figure 2.2: Proof of Lemma 1: Message chains during iterative construction.

previous iteration. In the later case *receive*$(m') \rightarrow$ *send*(m''). This argument can be continued resulting in a message chain

$$send(m') \rightarrow receive(m') \rightarrow send(m'') \rightarrow receive(m'') \ldots \rightarrow receive(m^0)$$

where *receive*(m^0) is in the initial state vector. (See Figure 2.2. The solid circles represent the initial state vector.)

We now show that the iteration terminates; suppose that it does not. At any iteration there are only finitely many inconsistent messages, since prefixes contain a finite number of receive events. Thus for the iteration to be non-terminating there must be an infinite message chain of the form described above. By the pigeonhole principle, in such a chain there must be two messages, say m_1 and m_2, that are sent by the same process. If $m_1 \rightarrow m_2$ in the chain then also $m_2 \rightarrow m_1$ because there is a local state which includes m_2 but not m_1. This cannot occur in any valid run. Therefore the iteration terminates.

Finally, we show that the local state of i is not changed, i.e. upon termination $c[i] = s^i_x$. Suppose that during the iteration state $c[i]$ is changed, due to a message m' sent by i. Again, there must be a chain of messages as above ending with *receive*(m^0) in the original states and *send*$(m') \rightarrow$ *receive*(m^0). There are two cases depending on what process has *receive*(m^0). (1) If it is process i then there is a circularity in $\rightarrow$ similar to the proof of termination above, as the message chain begins and ends with the same process. (2) Suppose that it is a process $j \neq i$. Recall that e^i_x is the last event of s^i_x and hence of the original $c[i]$. Recall also that, by the definitions of *MinSends*, the last event of j in the initial state vector is the sending of a message to i, call it m_j, which is received before or at e^i_x. We thus have that m_j is sent after *receive*(m^0) and received before or at e^i_x, so that *receive*$(m^0) \rightarrow e^i_x$. However, $e^i_x \rightarrow$ *send*(m') and *send*$(m') \rightarrow$ *receive*(m^0), resulting in an invalid circularity. Hence the final $c[i]$ is equal to s^i_j. ∎

In any state of the history of process i, i cannot determine which of the possible consistent cuts including its current state is an actual real-time global state, i.e. a set of local states that actually occur instantaneously during the execution. In this sense, a consistent cut is indistinguishable from a real-time global state. In defining epistemic concepts, the notion of indistinguishability plays a key role. For this reason we have chosen to use consistent cuts rather than real time in our logic for reasoning about asynchronous distributed systems.

Chapter 3

Distinguishable-Consistent-Cut Protocols

In this chapter we define a class of protocols called *distinguishable-consistent-cut protocols*, or *DCCPs*. DCCPs designate, in each run of a system, a set of local states composing a consistent cut. DCCPs are used in many distributed applications, including checkpointing [CL85,KT87], deadlock detection [BT87], distributed termination [Fra80], and broadcasting [BJ87].

DCCPs play a central role in the discussion of concurrent common knowledge in Chapter 4 and in the discussion of inhibition in Chapter 5. More specifically, in the next chapter we will prove that protocols of this class can be used to attain concurrent common knowledge of certain facts. In the following chapter we will examine the existence of DCCPs under a variety of system and protocol characteristics, including different types of inhibition.

The chapter is organized as follows. In Section 3.1, we formally define protocols and distinguishable-consistent-cut protocols. In Section 3.2, we give two DCCPs that are based on *flooding* algorithms [CL85,Cha82]. In Section 3.3, we give two DCCPs that are based on *two-phase* algorithms [BG81]. In the concluding section we begin a comparison of the differing characteristics of these protocols, which will be extended in Chapter 5.

3.1 Definitions

First, we give a formal definition of a *protocol* in our model. We define a protocol in an additive fashion, so as to separate the activities of the protocol from those of the underlying system; this will be useful later when we define inhibition so that we can compare system behavior with and without protocol intervention. We will require that any protocol allow the system to accomplish its original tasks as well as the protocol task. For example, we do not want to consider a protocol in which each process simply halts after sending its first message, without completing the necessary tasks of the underlying system. We will therefore require that the subsequence of system events in the run of a protocol be some valid run of the original system.

Our protocols will map a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, Act, En)$ to a new system $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, Act', En')$ under certain constraints. Instantiations of the actions in *Act* are called *system events* and instantiations of the actions in $Act' - Act$ are called *protocol events.* Given a local state consisting of both system and protocol events, the function *SysEvents* maps it to a new local state containing only the system events in their original order.

Definition 16. A *protocol* $\mathcal{P}$ is a function which maps a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, Act, En)$ to a new system $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, Act', En')$ such that (1) $Act \subseteq Act'$ (2) $En' = (En'_1, ..., En'_N)$, where En'_i is an enabling relation on i, and (3) for each total run $a' = (l'_1, ..., l'_N)$ of $\mathcal{P}(\mathcal{S})$, where l'_i is the local history of process i, run $a = (SysEvents(l'_1), ..., SysEvents(l'_N))$ is a total run of the original system $\mathcal{S}$.

The third condition says that the set of projections of possible protocol runs onto system events must be a subset of the original set of possible runs. In other words, a protocol cannot introduce new system behaviors; it can, however, eliminate some behaviors. For example, at points where the original protocol may have had a choice between performing internal events and send events, it may be forced to perform only internal events. Also, the underlying system can be forced to forego system activity for any number of protocol steps, as long as eventually a valid system run is completed. We consider such "inhibitions" in detail in Chapter 5.

For simplicity we consider protocols that distinguish only one consistent cut in each run of a system; our results regarding DCCPs can easily be extended to the case of multiple non-intersecting (periodic) cuts. We give three conditions on DCCPs. The first requires the designation of some consistent cut in every run, while the second requires that this cut be "distinguishable" by individual processes. The third condition implies that the DCCP cut may occur anywhere in the run of the underlying system; this eliminates DCCPs which, for example, only distinguish a cut at the beginning of a run.

Definition 17. A *distinguishable-consistent-cut protocol (DCCP)* for a system $\mathcal{S}$ is a protocol $\mathcal{P}$ in which (1) every run a of $\mathcal{P}(\mathcal{S})$ contains a unique consistent cut, denoted $cut(a)$, (2) for any runs a and a' of $\mathcal{P}(\mathcal{S})$ and any process i, if a' contains local state $cut(a)[i]$, then $cut(a)[i]$ is also $cut(a')[i]$, and (3) if a is a run of $\mathcal{S}$ and c is a consistent cut of a, then there is a run a' of $\mathcal{P}(\mathcal{S})$ which extends c, i.e. for each i, $c[i]$ is a prefix of $a'[i]$.

Given a particular run, a, of a DCCP and a particular process, i, we often use the term *cut state* to refer to $cut(a)[i]$, the local state of the process that is included in the DCCP cut.

The second condition of Definition 17 implies that the occurrence of the cut is deterministic with respect to the sequence of local events leading up to it, i.e. if process i's cut state immediately follows a sequence of events, s, in one run and exactly the same sequence, s, occurs in a different run, then i's cut state in the second run will immediately follow s also. In practice, this is necessary if some local action is to be immediately performed as a result of the occurrence of the cut; for example, taking a checkpoint. One can devise algorithms which determine the existence of a cut "after the fact" which do not satisfy condition (2)[GT89], i.e. a process may not "know" (in the next chapter we will formalize the notion of knowledge) that a particular local state is its cut state until well after the occurrence of the cut state in the local history. This may, however, be inappropriate or cause undesirable overhead in particular applications. In the checkpointing example, it necessitates taking multiple checkpoints, in order to encompass all of the local states which may later be included in the system checkpoint. This is undesirable due to the expense of taking a checkpoint. In the next chapter, it will be precisely the distinguishability characteristic that enables these protocols to attain concurrent common knowledge.

The third DCCP condition asserts that it is possible to distinguish a DCCP cut following any partial run of the original system. This eliminates, for example, a trivial protocol that designates the initial states as cut states.

We will sometimes consider DCCPs in which the designated cut is only consistent with respect to system messages—not necessarily with respect to protocol messages—as in the following definition.

Definition 18. A distinguishable-consistent-cut protocol *with respect to system messages* for a system $\mathcal{S}$ is a protocol $\mathcal{P}$ in which (1) every run $a = (l_1, \ldots, l_N)$ of $\mathcal{P}(\mathcal{S})$ contains a unique set of local states $cut(a) = (s_1, \ldots, s_N)$, such that $(SysEvents(s_1), \ldots, SysEvents(s_N))$ is a consistent cut of the run $(SysEvents(l_1), \ldots, SysEvents(l_N))$ of $\mathcal{S}$, (2) for any runs a and a' of $\mathcal{P}(\mathcal{S})$ and any process i, if a' contains local state $cut(a)[i]$, then $cut(a)[i]$ is also $cut(a')[i]$, and (3) if a is a run of $\mathcal{S}$ and c is a consistent cut of a, then there is a run a' of $\mathcal{P}(\mathcal{S})$ which extends c, i.e. for each i, $c[i]$ is a prefix of $a'[i]$.

Unless specifically stated, the original definition of DCCP will be assumed.

In all four of the protocols that follow, distinguishability is guaranteed because each cut state occurs immediately upon the completion of specific actions by the process. The third DCCP condition is guaranteed because the protocol actions may begin at any point in an initiator's local history. Thus the proofs of correctness for these protocols will be concerned only with the consistency of the cut states.

3.2 Flooding DCCPs

The DCCPs of this section are based on *flooding* algorithms. In a flooding algorithm, messages are sent along every channel in the system. Common examples of flooding algorithms are found in the checkpointing algorithm of Chandy and Lamport [CL85] and in the *echo* algorithms of Chang. (In echo algorithms, there are additional messages sent back along every channel, hence the name "echo.") We give two variants of this method, *Flood1* and *Flood2*. Both protocols require FIFO channels. Flood1, which is very similar to [CL85], is a DCCP with respect to system messages only; some of the protocol messages are inconsistent. In Flood2, both system and protocol messages are consistent.

In both Flood1 and Flood2, message propagation begins with a distinguished initiator I; we assume that an internal protocol event, denoted "StartDCCP," begins an initiator's execution of the protocol. Protocol messages are then sent to every process in the system, as each process forwards a protocol message to all of its neighbors. 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, the cut state is reached at the beginning of that interval. System messages sent to a neighbor after reaching the cut state and prior to sending the protocol message to that neighbor could be inconsistent; therefore sending such messages is not allowed. In Flood2, the cut state is reached at the end of the message-sending interval. System messages received while sending the protocol messages—thus before the cut state—could be inconsistent; in Flood2 receiving such messages is not allowed. (Protocol messages in Flood2 are not potentially inconsistent because they are all sent prior to a cut state. However, selectively receiving protocol messages but not system messages is not allowed in our model. Thus, all message receptions are disabled during the interval.)

Protocol 1. *Flood1.*

- The initiator I, at some point in its local history, executes StartDCCP, reaches its cut state and then sends $\langle I, j, Cut\rangle$ to all neighbors j. Between reaching the cut state and sending the protocol message to j, I sends no system messages to j.
- Each other process, i, upon first receiving a message of the form $\langle j, i, Cut\rangle$, reaches its cut state and then sends $\langle i, k, Cut\rangle$ to all neighbors $k \neq j$. Between reaching its cut state and sending the protocol message to k, i sends no system messages to k.

Theorem 1. *Flood1* is a distinguishable-consistent-cut protocol with respect to system messages for any reliable, FIFO system.

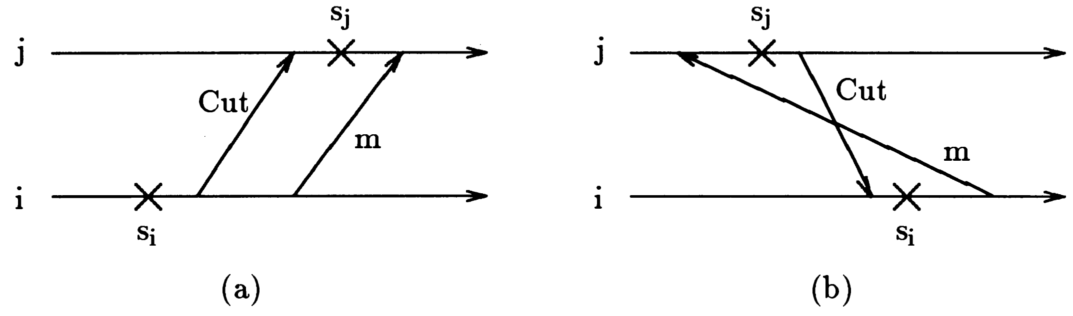

Figure 3.1: Proof of Theorem 1: (a) case 1, (b) case 2.

Proof: Consider any run of Flood1 in a reliable, FIFO system. For each process i, let s_i be the cut state indicated in the specification of the protocol. We show that states $SysEvents(s_1), SysEvents(s_2), \ldots, SysEvents(s_N)$ form a consistent cut by contradiction. Suppose to the contrary that there is a system message m sent after s_i but received before s_j. There are two cases to consider: (1) i sends a protocol message to j, or (2) i does not send a protocol message to j, which implies that i received its first protocol message from j.

In case (1), i must have sent $\langle i, j, Cut \rangle$ to j after s_i and before sending any system message to j, i.e. before sending m. Since channels are FIFO, $\langle i, j, Cut \rangle$ must reach j before message m. Unless j has already reached its cut state when receiving $\langle i, j, Cut \rangle$, it reaches the cut state upon receiving it. In either case it reaches s_j before receiving m. (See Figure 3.1 (a). Each "$\times$" denotes a cut state.) Therefore the assumption that m was received before s_j is false.

In case (2), since i receives its first protocol message from j, i reaches its cut state immediately upon reception of that message. Consequently, the protocol message is received before m is sent. Since j must send all protocol messages after reaching its cut state, m must be received by j before sending the protocol message to i. (See Figure 3.1 (b).) However, this implies causal circularity, since

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

Again the assumption that m was received before s_j must be false. ∎

Flood2 only differs from Flood1 in that both system and protocol messages are consistent, receives rather than sends are suspended temporarily, and the cut states occur after sending protocol messages rather than before. Their proofs of correctness are consequently very similar.

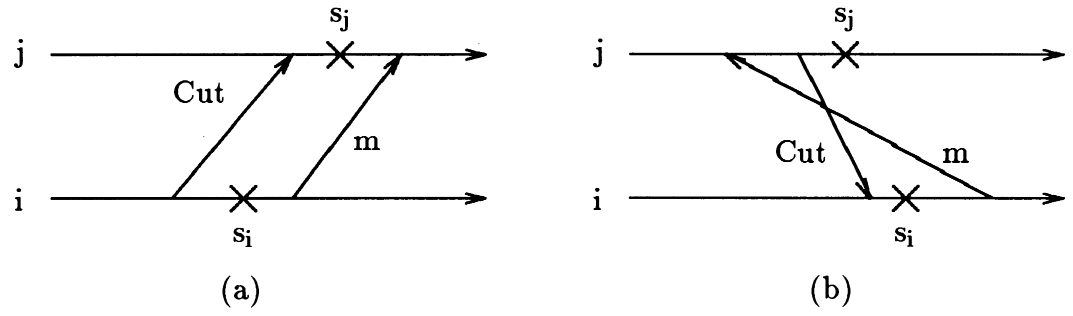

Figure 3.2: Proof of Theorem 2: (a) case 1, (b) case 2.

Protocol 2. *Flood2.*

- The initiator I, at some point in its local history, executes StartDCCP, sends $\langle I, j, Cut \rangle$ to all neighbors j, and then immediately reaches its cut state. Between sending the first protocol message and reaching the cut state, I receives no messages.
- Each other process, i, upon first receiving a message of the form $\langle j, i, Cut \rangle$, sends $\langle i, k, Cut \rangle$ to all neighbors $k \neq j$ and then immediately reaches its cut state. Between sending the first protocol message and reaching the cut state, i receives no messages.

Theorem 2. *Flood2* is a distinguishable-consistent-cut protocol for reliable, FIFO systems.

Proof: Consider any run of Flood2 in a reliable, FIFO system. For each process i, let s_i be the cut state indicated in the specification of the protocol. We show that states $s_1, s_2, \ldots, s_N$ form a consistent cut by contradiction. Suppose to the contrary that there is a message m sent after s_i but received before s_j. Note that m must be a system message, since cut states are not reached until all protocol messages have been sent. Again there are two cases to consider: (1) i sends a protocol message to j, or (2) i does not send a protocol message to j, which implies that i received its first protocol message from j.

In case (1), i must have sent m to j after s_i, by assumption, and consequently after sending $\langle i, j, Cut \rangle$ to j. Since channels are FIFO, the protocol message must reach j before message m. Unless j has already reached its cut state when receiving $\langle i, j, Cut \rangle$, it sends out its protocol messages and reaches its cut state before receiving any further messages. In either case, j reaches s_j before receiving m and the assumption that m was received before s_j is false. (See Figure 3.2 (a).)

In case (2), the protocol message from j arrives before s_i and consequently before the sending of m. By assumption m is received before s_j. Since j must send all protocol messages before reaching its cut state with no intervening receives, m must be received by j before sending the protocol message to i. (See Figure 3.2 (b).) However, this implies causal circularity, since

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

Again the assumption that m was received before s_j must be false. ∎

3.3 Two-Phase DCCPs

The next two DCCPs are *two-phase* protocols [BG81]; that is, there are two phases in which information is propagated from the initiator to each processor in the network. In both protocols, three sets of messages—labelled *Prepare, Cut,* and *Resume*—are sent respectively from the initiator to all processes, back to the initiator, and back to processes. The sending of non-protocol messages is suspended between non-initiators sending *Cut* and receiving *Resume.*

In Protocol 3, *Two-Phase*, we assume that messages between the initiator and other processes are forwarded as necessary on paths of length d or less, where d is the diameter of the network (recall that the network is not completely connected, so there may not be channels between the initiator and some processors); we will use the terminology that a process i "indirectly sends" a message to process j. In Protocol 4, *Tree*, messages are sent along a spanning tree of the network; consequently, it requires duplex channels so that messages may take the same path to and from the initiator.

Again, we assume that the event "StartDCCP" begins the initiator's protocol actions.

Protocol 3. *Two-Phase.*

1. The initiator I, at some point in its local history, executes StartDCCP and then indirectly sends $\langle I, j, Prepare\rangle$ to each process $j \neq I$.
2. Each process $j \neq I$, upon receiving $\langle I, j, Prepare\rangle$, disables sending of system messages, indirectly sends $\langle j, I, Cut\rangle$ to the initiator then reaches its cut state.
3. The initiator I, after receiving $\langle j, I, Cut\rangle$ from all processes $j \neq I$, immediately reaches its cut state and then indirectly sends $\langle I, j, Resume\rangle$ to all processes $j \neq I$.
4. Each process $j \neq I$, upon receiving $\langle I, j, Resume\rangle$, re-enables sending of system messages.

Theorem 3. Protocol 3, *Two-Phase*, is a distinguishable-consistent-cut protocol for reliable systems.

Proof: Consider any run of Two-Phase in a reliable system, and let state s_i be the cut state of process i, as indicated in the protocol. We show by contradiction that this set of states forms a consistent cut. Suppose to the contrary that message m is sent after one of these states but received before another. There are three cases to consider: (1) m is from I, to some $j \neq I$, (2) m is from some $j \neq I$, to I, and (3) m is from some i to some j, $i, j \neq I$.

In case (1), illustrated in Figure 3.3(a), let $m_1, ...m_k$ be the sequence of *Cut* messages from j to the initiator. The cut states of s_j and s_I immediately follow the sending of m_1 and the initiator's last reception of a *Cut* message from its children (not necessarily immediately after the reception of m_k), respectively. Since the inconsistent message m is received before s_j, it is received before *send*(m_1), which immediately precedes s_j. Clearly *send*(m_1) happens-before *receive*(m_k). But *receive*(m_k) is received by the initiator before s_I and hence before sending m. Thus the inconsistency of m results in a causal cycle, producing an invalid run.

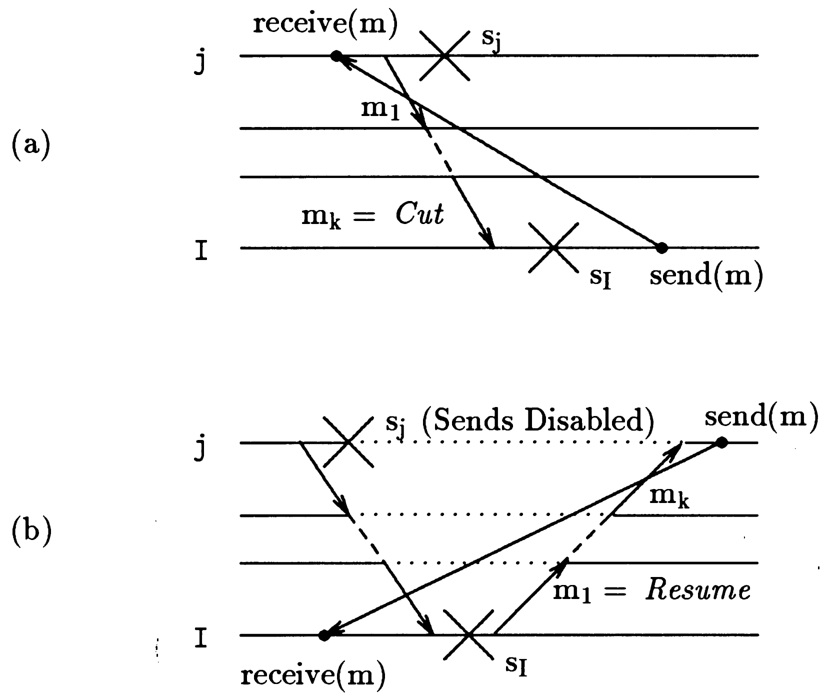

Figure 3.3: Proof of Theorem 3: (a) case 1, (b) case 2.

In case (2), illustrated in Figure 3.3(b), the sending of an inconsistent message m by j cannot occur until system sends are enabled, which happens after *Resume* is received. Let $m_1, ...m_k$ be the sequence of *Resume* messages from the initiator to j, so $receive(m_k)$ happens-before $send(m)$. If m is received before s_I then it is received before I sends m_1, and again an invalid circularity results.

Case (3), in which the inconsistent message m is between two non-initiators, is essentially a concatenation of case (1) and case (2). After the reception of m, j must begin a sequence of *Cut* messages to the initiator who then begins a sequence of *Resume* messages to i. Message m cannot be sent by i until after the resume, with the same result. Therefore the cut states are consistent. ∎

Finally, our fourth protocol, *Tree*, is an optimization of *TwoPhase* for the case in which channels are duplex. If this is the case, then communication can be limited to a spanning tree of the network. *Tree* is similar to protocols in [Fra80, Awe85].

Protocol 4. *Tree.*

1. 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.
2. The initiator I, at some point in its local history, executes StartDCCP and then sends *Prepare* to $children(I)$.
3. Each internal node i, after receiving *Prepare*, sends *Prepare* to $children(i)$.
4. Each leaf node i, after receiving *Prepare*, disables sending of system messages, sends *Cut* to $parent(i)$, and then reaches its cut state.
5. Each internal node i, after receiving *Cut* from $children(i)$, disables sending of system messages, sends *Cut* to $parent(i)$, and then reaches its cut state.
6. The initiator I, after receiving *Cut* from $children(i)$, reaches its cut state and then sends *Resume* to $children(I)$.
7. Each internal node i, after receiving *Resume* from $parent(i)$, sends *Resume* to $children(i)$ and then re-enables sending of system messages.
8. Each leaf node i, after receiving *Resume* from $parent(i)$, re-enables sending of system messages.

Theorem 4. Protocol 4, *Tree*, is a distinguishable-consistent-cut protocol for reliable systems with duplex channels.

We omit the proof of Theorem 4 as it is almost identical to that of Theorem 3.

3.4 Comparisons

Trade-offs between the four DCCPs discussed primarily involve: their message complexities, the degree to which they suspend activity of the underlying system, and requirements of FIFO channels. *TwoPhase* and *Tree* are more favorable in two of these respects: they do not require FIFO channels, and they are more message-efficient than the flooding protocols. *TwoPhase* uses only $3nd$ messages, where d is the diameter of the network, and *Tree* uses only only $3(n-1)$ messages. In contrast, *Flood1* and *Flood2* do require FIFO channels and use up to two messages per pair of neighboring processes, or $O(n^2)$ messages. However, *TwoPhase* and *Tree* disable system send events between their two phases; communication from the initiator is required to resume send activity, which in turn occurs only after the initiator receives communication from every process indirectly through its children. In contrast, *Flood1* and *Flood2* only suspend activity while a process is sending protocol messages to its neighbors. Hence they typically interfere less with the underlying system. The trade-off between the flooding protocols and the two-phase protocols depends on the degree that the system may be degraded by the suspension of activity. This suspension is termed *inhibition* and is addressed at length in Chapter 5.

Chapter 4

Concurrent Common Knowledge

In this chapter we define *concurrent common knowledge*. As noted previously, in doing so we replace the role of simultaneity in common knowledge with causal consistency; the resulting definition is attainable asynchronously and describes knowledge that actually occurs in several commonly-used protocols.

This chapter is organized as follows. In the first section we give a logic and semantic definitions for concurrent common knowledge and its components. In the second section we prove two important theorems regarding the attainability of CCK. The first theorem says that, if concurrent common knowledge is attained, it is attained along a single consistent cut of each run. The second theorem gives a general condition under which CCK of certain facts can be achieved; a corollary to that theorem establishes that DCCPs can be used to attain CCK of certain facts. In the third section we give a translation of our semantics to the more standard timed-runs semantics of [HM84]. In the final section we compare concurrent common knowledge to other proposed variants of common knowledge.

4.1 A Logic and Semantics for Concurrent Common Knowledge

The definition of concurrent common knowledge follows the standard pattern of defining a form of group knowledge, which we call "everyone concurrently knows," and then using a greatest fixed-point operator to define the appropriate variant of common knowledge [HM84]. Consequently, concurrent common knowledge of a fact ϕ will imply all formulas of the form "everyone concurrently knows ϕ and everyone concurrently knows that everyone concurrently knows ϕ" and so on.

In order to give a Kripkean interpretation of the knowledge modality, we need to identify an appropriate set of possible worlds and a family of possibility relations between those worlds. The discussion of concurrent knowledge really involves two modal operators and, hence, two different collections of accessibility relations in the semantics. This kind of situation is also seen in other variants of common knowledge. Discussions of eventual common knowledge and timestamped common knowledge [HM84,NT87] involve a temporal modality in addition to an epistemic modality.

4.1.1 A Logic for CCK

We will first introduce the symbols contained in our logic. Later we will define a formal semantics by stating when a formula is satisfied by a pair (a, c), where c is a consistent cut in asynchronous run a.

We assume that there is a set of primitive propositions *Prop*; these typically will be statements like "variable x in process i is 0" or "process i has sent a message m to process j". We represent these by lower-case letters $p, q, \ldots$.

We introduce two families of modal operators, each family indexed by process identifiers. They are written K_i and P_i respectively. Intuitively, $K_i(\phi)$ represents the statement "i knows ϕ," which in terms of asynchronous systems means "ϕ is true in all possible consistent global states that include i's local state." The formula $P_i(\phi)$ represents the statement "there is some consistent global state in this run, including i's local state, in which ϕ is true." P has, roughly speaking, a role similar to a temporal modality. It is quite different, however, from the familiar temporal operators like $\Diamond$. Axioms obeyed by K and P are given in Section 4.1.3.

The next modal operator is written E^C and stands for "everyone concurrently knows." This is weaker than E, which stands for "everyone knows (now)." The definition of $E^C(\phi)$ is

$$E^C(\phi) \Leftrightarrow \bigwedge_{i \in Proc} K_i P_i(\phi)$$

The last modal operator that we introduce is C^C, concurrent common knowledge. Analogously to common knowledge, we wish to define a state of processor knowledge which implies that all processors are in that same state of knowledge, with respect to ϕ, along some cut of the run. In other words, we want a state of knowledge X satisfying

$$X = E^C(\phi \wedge X).$$

Thus we want concurrent common knowledge to be a fixed point of $E^C(\phi \wedge X)$. C^C will be defined semantically as the weakest such fixed point, namely the greatest fixed point of $E^C(\phi \wedge X)$. It therefore satisfies

$$C^C(\phi) \Leftrightarrow E^C(\phi \wedge C^C(\phi))$$

and

$$C^C(\phi) \Rightarrow E^C(\phi) \wedge (E^C)^2(\phi) \wedge (E^C)^3(\phi) \ldots,$$

i.e. that $(E^C)^k \phi$ holds for any k. The greatest fixed-point definition is, however, stronger than the infinite conjunction.

4.1.2 Formal Semantics of CCK

In an asynchronous system $\mathcal{S}$, the possible worlds are the consistent cuts of the set of possible asynchronous runs $\mathcal{A}(\mathcal{S})$. We use pair (a, c) to stand for the consistent cut c in the asynchronous run a. Recall that a cut is an N-vector of local states, one for each process in *Proc*. Two cuts are viewed as indistinguishable by process i if they contain the same local state of process i. This is clearly an equivalence relation.

Definition 19. We write $(a, c) \sim_i (a', c')$ to represent the indistinguishability of (a, c) and (a', c') to i:

$$(a, c) \sim_i (a', c') \Leftrightarrow (a, c)[i] = (a', c')[i]$$

The formal semantics is given via the definition of the satisfaction relation $\models$. We assume that we are given a function π that assigns a truth value to each primitive proposition p and local state s of process i. The truth of a primitive proposition p in (a, c) is determined by π and c. This defines $(a, c) \models p$. The satisfaction relation is defined in the obvious way for formulas built up using the logical connectives. The following defines the meaning of K_i in our setting; throughout this section, the range of quantification is $\mathcal{A}(\mathcal{S})$:

$$(a, c) \models K_i(\phi) \Leftrightarrow \forall (a', c') \ : \ (a', c') \sim_i (a, c) \ : \ (a', c') \models \phi$$

This is practically the same as the definition in Halpern and Moses [HM84], except that we use asynchronous runs rather than timed runs. This definition also validates the standard axioms for knowledge, which we reiterate later in this section.

The meaning of P_i is given by the following definition

$$(a, c) \models P_i(\phi) \Leftrightarrow \exists (a, c') \ : \ (a, c') \sim_i (a, c) \ : \ (a, c') \models \phi$$

In other words, $P_i(\phi)$ states that there is some cut, *in the same asynchronous run*, including i's current state, such that ϕ is true in that cut. (See Figure 4.1.) Another way of viewing the meaning of P_i is to define the equivalence relation $\approx_i$ to stand for indistinguishable cuts in the same run; this is a refinement of the $\sim_i$ relation.

$$(a, c) \approx_i (a', c') \Leftrightarrow (a = a') \wedge (a, c) \sim_i (a', c')$$

Given the definition of $\approx_i$ we can equivalently define P_i as

$$(a, c) \models P_i(\phi) \Leftrightarrow \exists (a', c') \ : \ (a', c') \approx_i (a, c) \ : \ (a', c') \models \phi.$$

The remainder of our formal semantics outlines the definition of C^C using greatest fixed points. In order to define the meaning of C^C using fixed points we need to define the meaning of formulas with free variables, written $X, Y \ldots$,

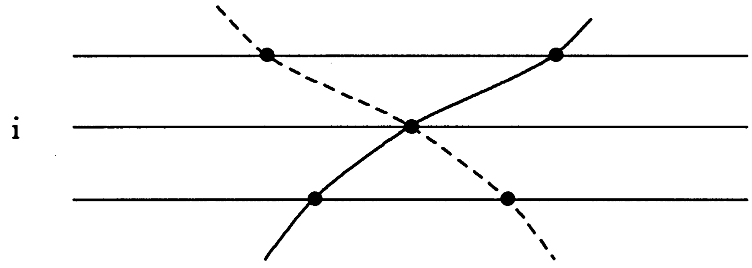

Figure 4.1: Satisfaction of $(a, c) \models P_i(\phi)$.

in them. We think of the meaning of such a formula as a function from sets of consistent cuts to sets of consistent cuts. Let us call the set of consistent cuts W. Then we can define the following meaning function for all formulas. In the following we let Z stand for a generic subset of W. The meaning of the formulas is given by the inductively defined function $\mathcal{M}$. The meaning of the primitive propositions $p, q, \ldots$ is given by a function π as discussed above. The meaning function defined below follows, very closely, the definition given by Halpern and Moses [HM84,Koz83].

1. $\mathcal{M}[\![p]\!](Z) = \{u \in W | \pi(u, p) = \mathbf{true}\}$ where p is a primitive proposition.
2. $\mathcal{M}[\![\neg\phi]\!](Z) = W - \mathcal{M}[\![\phi]\!](Z)$.
3. $\mathcal{M}[\![\phi \wedge \psi]\!](Z) = \mathcal{M}[\![\phi]\!](Z) \cap \mathcal{M}[\![\psi]\!](Z)$.
4. $\mathcal{M}[\![X]\!](Z) = Z$
5. $\mathcal{M}[\![K_i(\phi)]\!](Z) = \{(a, c) \in W | \forall (a', c') \in W : (a, c) \sim_i (a', c') \Rightarrow (a', c') \in \mathcal{M}[\![\phi]\!](Z)\}$.
6. $\mathcal{M}[\![P_i(\phi)]\!](Z) = \{(a, c) \in W | \exists (a', c') \in W : (a, c) \approx_i (a', c') \Rightarrow (a', c') \in \mathcal{M}[\![\phi]\!](Z)\}$.

If a formula does not contain a free variable then its meaning is a constant function. The truth value definition of the semantics can be recovered by defining

$$(a, c) \models \phi \text{ iff } (a, c) \in \mathcal{M}[\![\phi]\!](\emptyset).$$

In fact the semantic clauses just given are exactly what one would expect for the Tarski-style truth definition except that they have been given in terms of sets and set operations instead of truth values and logical connectives.

We will define C^C via a greatest fixed-point operator. We extend the syntax by $\nu X.\phi$. The interpretation of this proceeds as follows. The existence of the greatest fixed point depends upon the *monotonicity* of $\mathcal{M}[\![\phi]\!](Z)$; a function f is monotonic if $A \subseteq B$ implies $f(A) \subseteq f(B)$. To guarantee monotonicity, We require that free occurrences of X in ϕ be *positive*, i.e. all occurrences of X are in the scope of an even number of negation signs. This is clearly a syntactic property. It is easy to see (by induction on the structure of formulas) that $\mathcal{M}[\![\phi]\!](Z)$ will be a monotonic function if X appears positively. Any monotonic function on a complete lattice has a greatest fixed point [Tar55]. The powerset 2^W ordered by inclusion is a complete lattice. We can thus give meaning to $\nu X.\phi$ as

$$\mathcal{M}[\![\nu X.\phi]\!](Z) = \cup\{B | \mathcal{M}[\![\phi]\!](B) = B\}$$

and $C^C(\phi)$ can be viewed as a special case of this as follows

$$C^C(\phi) = \nu X.E^C(\phi \wedge X)$$

It is not true that $E^C(\phi \wedge X)$ defines a continuous function so the fixed point is not attained by simply iterating up to ω; in this sense C^C is rather like $C^\diamond$ [HM84]. Note that in the logic the only occurrence of greatest fixed points is through occurrences of C^C. In such occurrences one never has to interpret formulas like $\nu X.\nu Y.X \wedge Y$ where there are fixed points of expressions containing more than one free variable.

The fact that concurrent common knowledge is a greatest fixed point is expressed by an induction rule. The following theorem gives this rule and establishes its soundness.

Theorem 5. The following induction rule is sound with respect to the semantics defined above. If $\phi \Rightarrow E^C\phi$ is valid then so is $\phi \Rightarrow C^C\phi$.

Proof: Assume that $\phi \Rightarrow E^C\phi$. Semantically, this means that $\mathcal{M}[\![\phi]\!](\emptyset) \subseteq \mathcal{M}[\![E^C\phi]\!](\emptyset)$. Recall, the meaning of $C^C\phi$ is the greatest fixed point of the functional $F = \lambda u.\mathcal{M}[\![E^C(\phi \wedge X)]\!](u)$, where u is an element of W. Now

$$\begin{aligned}
&F(\mathcal{M}[\![\phi]\!](\emptyset)) = \mathcal{M}[\![E^C(\phi \wedge X)]\!](\mathcal{M}[\![\phi]\!](\emptyset)) \\
&= \mathcal{M}[\![E^C\phi]\!](\mathcal{M}[\![\phi]\!](\emptyset)) \cap \mathcal{M}[\![E^C X]\!](\mathcal{M}[\![\phi]\!](\emptyset)) \\
&= \mathcal{M}[\![E^C\phi]\!](\emptyset).
\end{aligned}$$

The first equality is by definition of F; the second and third are by explicit calculation, using the semantic definitions. Thus, by our assumption, $\mathcal{M}[\![\phi]\!](\emptyset) \subseteq F(\mathcal{M}[\![\phi]\!](\emptyset))$. Since F is monotonic, we have the following chain of inclusions

$$\mathcal{M}[\![\phi]\!](\emptyset) \subseteq F(\mathcal{M}[\![\phi]\!](\emptyset)) \subseteq F(F(\mathcal{M}[\![\phi]\!](\emptyset)))\ldots$$

Thus $\mathcal{M}[\![\phi]\!](\emptyset) \subseteq \cup_\alpha F^\alpha(\mathcal{M}[\![\phi]\!](\emptyset))$, where α ranges through the ordinals. Now $\cup_\alpha F^\alpha(\mathcal{M}[\![\phi]\!](\emptyset))$ is a fixed point of F, hence, $\cup_\alpha F^\alpha(\mathcal{M}[\![\phi]\!](\emptyset)) \subseteq \mathcal{M}[\![C^C\phi]\!]$. Thus we get $\mathcal{M}[\![\phi]\!](\emptyset) \subseteq \mathcal{M}[\![C^C\phi]\!]$; which establishes the soundness of the induction rule.∎

4.1.3 An Axiom System

We conclude this section with a collection of axioms obeyed by K and P. These are:

1. all instances of propositional tautologies
2. $K_i(\phi) \Rightarrow \phi$
3. $K_i(\phi) \wedge K_i(\phi \Rightarrow \psi) \Rightarrow K_i(\psi)$
4. $K_i(\phi) \Rightarrow K_i(K_i(\phi))$
5. $\neg K_i(\phi) \Rightarrow K_i(\neg K_i(\phi))$
6. $\phi \Rightarrow P_i(\phi)$
7. $P_i(P_i(\phi)) \Rightarrow P_i(\phi)$
8. $P_i(\neg P_i(\neg\phi)) \Rightarrow P_i(\phi)$
9. $P_i(\phi) \Rightarrow \neg K_i(\neg\phi)$

The first five axioms are the usual axioms for knowledge [Hin62] and are known to be a complete axiomatization of the possible worlds semantics once we add the usual rules of inference, namely *modus ponens* and knowledge generalization. It is easy to see that these axioms are sound for the semantics given in this section. The soundness of Axiom 6 follows from the definition. Recall that $(a, c) \models P_i\phi$ if there exists an (a, c'), $(a, c') \sim_i (a, c)$, such that $(a, c') \models \phi$. If we have $(a, c) \models \phi$ we can choose c' to be c and we immediately have $(a, c) \models P_i\phi$. The soundness of the other axioms follow equally easily.

Note that it does not follow from the axioms that

$$C^C(\phi \Rightarrow \psi) \wedge C^C(\phi) \Rightarrow C^C(\psi)$$

or that $E^C(\phi) \Rightarrow \phi$. This is not necessary for our purposes. Note that the truth of $E^C(\phi)$ is determined with respect to some cut (a, c). A process cannot distinguish which cut, of the perhaps multiple cuts that are in the run and consistent with its local state, satisfies ϕ, i.e. whether or not c does; it can only know the existence of such a cut.

4.2 Attaining Concurrent Common Knowledge

In this section we first prove two theorems regarding the attainment of concurrent common knowledge. First, we show that if CCK is attained in a run then all processes learn it along a single consistent cut of the run. Secondly, we prove a general condition under which systems attain CCK. As a corollary to that theorem, we show that distinguishable-consistent-cut protocols attain CCK under certain conditions. In the final section we give an example DCCP variant for attaining CCK.

4.2.1 Single-Cut Theorem

For ordinary common knowledge, C, it is a theorem that if C is attained then all processes learn it simultaneously[HM84,PR85]. An analogous theorem holds for concurrent common knowledge. Before stating the theorem, we will first formalize the notions of "attainment" and "learning."

Definition 20. A fact ϕ is *attained* in run a if $\exists c \,:\, (a,c) \models \phi$.

Likewise, we say that a system attains ϕ if every run of the system attains ϕ.

In this section and the following section we will often refer to "knowing" a fact in a state rather than in a consistent cut. Recall that knowledge is dependent only on the local state of a process, since $(a,c) \sim_i (a',c')$ iff $(a,c)[i] = (a',c')[i]$; therefore such terminology is reasonable. Formally, *i knows ϕ in state s* is shorthand for

$$\forall(a,c) \,:\, (a,c)[i] = s \,:\, (a,c) \models \phi.$$

Definition 21. Process *i learns ϕ in state s^i_j* of run a if i knows ϕ in s^i_j and, for all states s^i_k in run a, $k < j$, i does not know ϕ.

The following theorem says that if $C^C(\phi)$ is attained in a run then all processes i learn $P_iC^C(\phi)$ along a single consistent cut.

Theorem 6. If $C^C(\phi)$ is attained in a run a, then the set of states in which each process i learns $P_iC^C(\phi)$ forms a consistent cut in a.

Proof: $C^C(\phi)$ is attained in a implies that there exists a consistent cut where $C^C(\phi)$ holds. Since $C^C(\phi) \Rightarrow \wedge_i K_iP_iC^C(\phi)$, there must exist states $s_1, \ldots, s_N$ such that s_i is the state in which i learns $P_iC^C(\phi)$. We will show that cut $c = \langle s_1, \ldots, s_N\rangle$ must be consistent.

Suppose that c is inconsistent. Then there must be a message m, say from process j to process k, such that *receive*(m) is included in state s_k but *send*(m) is not included in state s_j. Any *consistent* cut c' where $(a,c')[j] = s_j$ cannot include the reception of m, since s_j does not include *send*(m). Furthermore, by the definition of s_k, the reception of m occurs before k learns $P_kC^C(\phi)$. Therefore in any consistent cut c' where $(a,c')[j] = s_j$, k does not know $P_kC^C(\phi)$. We next show that this is impossible.

By the definition of s_j, j knows $P_jC^C(\phi)$ in s_j, i.e. in any consistent cut c_1 where $(a,c_1)[j] = s_j$, $(a,c_1) \models P_jC^C(\phi)$. By the definition of P, this means that there is some consistent cut c_2, $(a,c_2)[j] = (a,c_1)[j] = s_j$, for which $(a,c_2) \models C^C(\phi)$. The definition of C^C implies furthermore that $(a,c_2) \models \wedge_{i\in Proc}K_iP_iC^C(\phi)$ which in turn implies $(a,c_2) \models K_kP_kC^C(\phi)$. This contradicts the statement above that, in any consistent cut c' where $(a,c')[j] = s_j$ (including (a,c_2)), k does not know $P_kC^C(\phi)$. Hence the supposition that c is inconsistent must be false, making the theorem true. ∎

It is this theorem that illustrates an important difference between $C(\phi)$ and $C^C(\phi)$. In asynchronous systems, simultaneous action of any kind is impossible. Action coordinated to occur along a consistent cut is, however, easily achievable. Consequently, in the following sections we will be able to illustrate that CCK is attainable in asynchronous distributed systems.

4.2.2 Attainment Theorem

We proceed next by presenting our Attainment Theorem, which gives a general criterion by which concurrent common knowledge may be attained in distributed protocols. It is then a corollary to this theorem that DCCPs attain concurrent common knowledge under certain conditions.

In order to achieve $C^C(\phi)$, it will be sufficient that a system have a set $\mathcal{D}$ of cuts, at least (but not exactly) one per run, with the distinguishability property as for DCCPs, i.e. when the local state of any process is in a cut of $\mathcal{D}$ in some run, then the same local state of that process is at some cut of $\mathcal{D}$ in every run in which it occurs. The implies that the process *knows* that it is at one of the cuts. We describe this semantically by defining *locally-distinguishable cut sets* below.

Definition 22. A *locally-distinguishable cut set* $\mathcal{D}$ of a system $\mathcal{S}$ is a set of cuts $\mathcal{D}$ such that

$$\forall a \in \mathcal{S} \ : \ \exists (a, c) \in \mathcal{D}$$

and

$$\forall i \in Proc \ : \ \forall (a, c) \in \mathcal{D} \ :$$
$$\forall (a', c') \sim_i (a, c) \ : \ \exists (a', d) \in \mathcal{D} \ : \ (a', d) \sim_i (a, c).$$

In the definition above, suppose that we let $in\mathcal{D}$ stand for a formula such that $(a, c) \models in\mathcal{D}$ iff $(a, c) \in \mathcal{D}$. Given the formula $in\mathcal{D}$, the second condition for a locally-distinguishable cut set can be rewritten as simply

$$in\mathcal{D} \Rightarrow E^C(in\mathcal{D}).$$

We now show the primary result of this section: any system that guarantees that there is a locally-distinguishable cut set where a fact ϕ holds attains concurrent common knowledge of ϕ.

Theorem 7. **(Attainment Theorem)** If a system $\mathcal{S}$ has a locally-distinguishable cut set $\mathcal{D}$ such that

$$\forall (a, c) \in \mathcal{D} : (a, c) \models \phi$$

then

$$\forall (a, c) \in \mathcal{D} : (a, c) \models C^C(\phi),$$

i.e. the system attains concurrent common knowledge of ϕ.

Proof: Let $in\mathcal{D}$ stand for a formula such that $(a,c) \models in\mathcal{D}$ iff $(a,c) \in \mathcal{D}$, as above. By the definition of a locally-distinguishable cut set, $in\mathcal{D} \Rightarrow E^C(in\mathcal{D})$. By the conditions of the theorem, $in\mathcal{D} \Rightarrow \phi$. Note that, since $in\mathcal{D} \Rightarrow \phi$, $\phi \wedge in\mathcal{D}$ is equivalent to $in\mathcal{D}$. Thus we have

$$\phi \wedge in\mathcal{D} \Rightarrow E^C(\phi \wedge in\mathcal{D}).$$

Then, by the induction axiom,

$$\phi \wedge in\mathcal{D} \Rightarrow C^C(\phi \wedge in\mathcal{D}).$$

It follows from the semantic definitions and the validity of $\phi \wedge in\mathcal{D} \Rightarrow \phi$ that $C^C(\phi \wedge in\mathcal{D}) \Rightarrow C^C(\phi)$. Thus we have that $in\mathcal{D} \Rightarrow C^C(\phi)$, or equivalently $\forall(a,c) \in \mathcal{D} : (a,c) \models C^C(\phi)$. ∎

The following corollary extends this result to the fact that DCCPs attain concurrent common knowledge if they guarantee that the fact ϕ holds on each $cut(a)$ in the definition of DCCP. The proof is trivial, letting the set $\mathcal{D}$ above be

$$\{(a,c) | a \in \mathcal{A}(\mathcal{P}(\mathcal{S})) \wedge c = cut(a)\}.$$

Note that DCCPs are a special case of protocols that satisfy the theorem, since they create exactly one cut per run; the theorem applies to an arbitrary cut set $\mathcal{D}$ (even one containing intersecting cuts).

Corollary 1. Any DCCP that, for all runs a, guarantees $(a, cut(a)) \models \phi$ attains $C^C(\phi)$.

4.2.3 Attaining CCK via DCCPs

In this section we give an example of using a DCCP to attain concurrent common knowledge.

We must first consider for which facts ϕ attaining $C^C(\phi)$ is possible. Certainly $C^C(\phi)$ cannot be attained by a DCCP if ϕ is false or will be falsified during execution of the protocol. We say that ϕ is *locally controllable by* i if, whenever i knows ϕ in any state, i can prevent falsifying ϕ for any finite number of events. Note that any *stable* fact—one that, once true, remains true forever—is always locally controllable by any process. Unstable facts concerning the local state of i are also usually locally controllable by i. An example of this is $x_i = 1$, where x_i is a local variable of i. Any fact that becomes known to some process and is locally controllable by that process can become concurrent common knowledge of all processes.

The correctness of the following protocol, a modification of Flood2 from Chapter 3, requires that the fact ϕ be locally controllable by the initiator I. We modify Flood2 to insure that ϕ holds during the cut state. To do this, we require that

ϕ be known to be true initially and prevent falsifying it until the cut states have been reached. Note that in order to have a locally-distinguishable-cut set, which requires that there be a locally-distinguishable cut in every run, ϕ must become known at some point in all runs. Alternatively, we could weaken our definitions so that only runs in which ϕ becomes known to a process must contain an element of the locally-distinguishable cut set. For simplicity we use the current scheme.

Protocol 5. *Flood2'*, attainment of $C^C(\phi)$.

- The initiator I, at some point in its local history where I knows ϕ, sends $\langle I, j, (\phi, CCK)\rangle$ to all neighbors j and then immediately reaches its cut state. Between sending the first message and reaching the cut state, I receives no messages and prevents falsifying ϕ.
- Each other process, i, upon first receiving a message of the form $\langle j, i, (\phi, CCK)\rangle$, sends $\langle i, k, (\phi, CCK)\rangle$ to all neighbors $k \neq j$ and then immediately reaches its cut state. Between sending the first message and reaching the cut state, i receives no messages.

Theorem 8. Let $\mathcal{S}$ be a system with reliable, FIFO channels in which ϕ is locally controllable by I and in which ϕ becomes known to I. Then *Flood2'*$(\mathcal{S})$ attains $C^C(\phi)$.

Proof: Let $\mathcal{D}$ be the set of possible consistent cuts characterized as in Theorem 2. Since the protocol always forms exactly one of these cuts, and the cut state of each process always immediately follows the sending of the protocol messages, $\mathcal{D}$ is a locally-distinguishable cut set. Since ϕ holds initially and on any cut up through I's completion of the protocol, $\forall (a,c) \in \mathcal{D} : (a,c) \models \phi$. The theorem then follows from Theorem 7. ∎

The method above, restricting the initiation of a DCCP to a point in the run where it is known to the locally-controlling initiator, trivially extends to the other DCCPs presented in the previous chapter.

In sum, we have shown in this section that concurrent common knowledge is attainable in asynchronous systems by using DCCPs. This makes CCK a potentially useful form of knowledge, as it describes states that can and do arise in such systems. Furthermore, we have shown in the previous chapter that there are simple and efficient DCCPs. Thus, given a problem that can be formulated in terms of C^C, a simple and efficient solution immediately follows from these results.

4.3 Translation to Timed Runs Semantics

Timed runs have been used by Halpern and Moses to provide formal semantics for common knowledge and the other variants of common knowledge that they introduce. For our purposes, the asynchronous runs provide a better choice since time does not enter our system model. Nevertheless, it has become standard to use timed runs to model a variety of different systems; we will show that our logic can be given a timed-runs semantics as well. Furthermore, we show that the semantics in terms of timed runs is essentially equivalent to the asynchronous runs semantics.

The translation proceeds by first defining an appropriate set of possible timed runs R and a primitive proposition function π_R, given an initial set of possible asynchronous runs A and a primitive proposition function π_A. Next, we define our new modal operators using timed runs semantics. Finally, we state a theorem which formally relates our asynchronous semantics to the timed runs semantics, and prove it by structural induction on the formulas in the logic.

4.3.1 The Set of Possible Timed Runs

We first give a definition for timed runs using our notation that is consistent with that of [HM84].

Definition 23. A *timed run* r is a sequence of N-vectors of local states (or equivalently, event or action sequences) indexed by a possibly-infinite sequence of natural numbers, such that, for each processor i, $(r,t)[i]$ is a prefix of $(r,t+1)[i]$.

We associate an asynchronous run a with $timing(a)$, the set of all timed runs having the same events and causal structure. To preserve causal structure, we will require that the real-time values associated with events be consistent with the happens-before relation. Let $time(r,e)$ be the time value t of the latest global state (r,t) preceding the occurrence of event e.

Definition 24. Given a run r and event e in $r[i]$, $time(r,e)$ is the natural number t such that $e \notin (r,t)[i] \wedge e \in (r,t+1)[i]$.

Definition 25. Given an asynchronous run a, $timing(a)$ is the set of all timed runs r such that:

1. A state is in r iff it is in a, i.e.

$$\forall t : \forall i : \exists c : (r,t)[i] = (a,c)[i] \text{ and } \forall c : \forall i : \exists t : (a,c)[i] = (r,t)[i].$$

2. Causal structure of events is preserved, i.e. for all events e_1 and e_2, if $e_1 \rightarrow e_2$ then $time(e_1) < time(e_2)$.

Note that for any one asynchronous run there are, in general, infinitely many corresponding timed runs. However, there is exactly one asynchronous run corresponding to each timed run, because a timed run has exactly one causal structure. Now we can define the set of timed runs to be the union of all asynchronous timings, so that $R \equiv \cup_{a \in A} timing(a)$.

We next prove that every real-time global state has a corresponding consistent cut. Note that we can assert the equality of real-time global states and consistent cuts since they are both N-vectors of local states.

Theorem 9. Given a run r in $timing(a)$, $\forall t : \exists! c : (r,t) = (a,c)$.

Proof: The proof is by contradiction. By condition (1) in the definition of $timing(a)$, every local state of r is a local state of a. Therefore for the theorem to be false the local states of (r,t) must be inconsistent. Thus run r must contain events e_1 and e_2 such that $e_1 \rightarrow e_2$, e_2 is contained in (r,t), and e_1 is not contained in (r,t). Since e_1 is in r but not in (r,t), $time(r,e_1) \geq t$ and similarly $time(r,e_2) < t$. However, this contradicts condition (2) of the definition of $timing(a)$, which requires that if $e_1 \rightarrow e_2$ then $time(r,e_1) < time(r,e_2)$. Uniqueness follows because a set of local states uniquely specifies a cut in an asynchronous system. ∎

Due to the preceding theorem, we can define $Cut(r,t)$ to be the consistent cut corresponding to real-time global state (r,t).

Definition 26. $Cut(r,t)$ is the pair (a,c) such that r is in $timing(a)$ and $(r,t) = (a,c)$.

It is also true that for any (a,c) there is some (r,t) such that $Cut(r,t) = (a,c)$. This is a straightforward consequence of the definitions of consistent cuts and $timing(a)$; we leave the proof of the theorem to the reader.

Theorem 10. $\forall (a,c) : \exists (r,t) : Cut(r,t) = (a,c)$

We next define the primitive proposition function π_R from the primitive proposition function π_A in the obvious way.

Definition 27. $\pi_R((r,t),\phi) \equiv true$ iff $\pi_A(Cut(r,t),\phi) = true$.

4.3.2 Timed-Runs Semantics

The timed-runs semantics for the ordinary logical connectives, the primitive propositions, the knowledge modality K_i, and the greatest fixed-point operator are exactly as in Halpern and Moses [HM84]. In this subsection we only discuss the new modal operator, P_i. In order to avoid confusion, we use the symbols $\models_T$ and $\models_A$ to stand for timed-run semantics and asynchronous-runs semantics, respectively.

First we define the relation $\bowtie$ on timed runs, to represent timed runs that are timings of the same asynchronous run.

Definition 28. Given timed runs r and r', $r \bowtie r'$ iff there exists an asynchronous run a such that $r \in timing(a)$ and $r' \in timing(a)$.

Clearly (r', t') is composed of local states forming a consistent cut of r if $r \bowtie r'$. This motivates the following definition for the meaning of P_i in a timed-runs semantics.

Definition 29.

$$(r,t) \models_T P_i(\phi) \Leftrightarrow \exists(r',t') : (r' \bowtie r) \wedge ((r',t') \sim_i (r,t)) : (r',t') \models \phi$$

The definition of E^C is derivable from the definitions of $\wedge$, K_i, and P_i and is as follows:

$$(r,t) \models_T E^C\phi \Leftrightarrow \forall(r',t') : (r',t') \sim_i (r,t) :$$
$$\exists(r'',t'') : (r'' \bowtie r') \wedge ((r'',t'') \sim_i (r',t')) : (r'',t'') \models_T \phi$$

Note that $(r,t) \models_T E^C\phi$ is weaker than $(r,t) \models_T E\phi$, since processes know that ϕ is true of some consistent cut intersecting (r,t) rather than of (r,t) itself. This is essentially why we can asynchronously attain the greatest fixed point of E^C but not of E. Finally, $(r,t) \models_T C^C\phi$ is defined using the same greatest fixed-point interpretation as in [HM84] and as in our asynchronous runs semantics.

4.3.3 The Equivalence of the Two Semantics

Two preliminary facts are trivial consequences of our definitions.

Fact 1. Let $(a,c) = Cut(r,t)$ and $(a',c') = Cut(r',t')$. Then

(a) $(a,c) \sim_i (a',c')$ iff $(r,t) \sim_i (r',t')$, and
(b) $a = a'$ iff $r \bowtie r'$.

The following theorem relates the truth of formulas in our asynchronous-runs semantics to the truth of formulas in the timed-runs semantics.

Theorem 11. $(r,t) \models_T \phi \Leftrightarrow Cut(r,t) \models_A \phi$

Proof: By structural induction on formulas.

1. Primitive propositions:
   Follow immediately from definition of π_r.
2. $\neg, \wedge$: From the structural induction hypothesis,
   $(r,t) \models_T \phi \Leftrightarrow Cut(r,t) \models_A \phi$, it follows immediately that
   $(r,t) \models_T \neg\phi \Leftrightarrow Cut(r,t) \models_A \neg\phi$ and
   $(r,t) \models_T \phi \wedge \psi \Leftrightarrow Cut(r,t) \models_A \phi \wedge \psi$.
3. K_i: $(\Rightarrow)$ It is given that $(r,t) \models_T K_i\phi$, i.e. that

$$\forall(r',t') : (r',t') \sim_i (r,t) : (r',t') \models_T \phi.$$

By Theorem 10,

$$\forall(a,c):(a,c)\sim_i Cut(r,t):\exists(r'',t''):Cut(r'',t'')=(a,c).$$

Furthermore, Fact 1(a) implies that $(r'',t'')\sim_i(r,t)$ and since $(r,t)\models_T K_i\phi$, $(r'',t'')\models_T \phi$. By the structural induction hypothesis we then have that $(a,c)\models_A \phi$, therefore $Cut(r,t)\models_A K_i\phi$ as desired.

($\Leftarrow$) It is given that $Cut(r,t)\models_A K_i\phi$, i.e. that

$$\forall(a,c):(a,c)\sim_i Cut(r,t):(a,c)\models_A \phi.$$

From Fact 1(a),

$$\forall(r',t'):(r',t')\sim_i(r,t):Cut(r',t')\sim_i Cut(r,t).$$

Since $Cut(r,t)\models_A K_i\phi$ it follows that

$$\forall(r',t'):(r',t')\sim_i(r,t):Cut(r',t')\models_A \phi.$$

Then $(r',t')\models_T \phi$ follows from the structural induction hypothesis, hence $(r,t)\models_T K_i\phi$.

4. P_i: ($\Rightarrow$) It is given that $(r,t)\models_T P_i\phi$, i.e.

$$\exists(r',t'):(r'\bowtie r)\wedge(r',t')\sim_i(r,t):(r',t')\models_T \phi.$$

Let $(a,c)=Cut(r,t)$ and $(a',c')=Cut(r',t')$. Then by Fact 1, $a=a'$ and $(a,c)\sim_i(a',c')$. Therefore

$$\exists(a,c'):(a,c')\sim_i(a,c):(a,c')=Cut(r',t')\wedge(r',t')\models_T \phi.$$

$(a,c')\models_A \phi$ by the induction hypothesis, so that $(a,c)\models_A P_i\phi$.

($\Leftarrow$) It is given that $Cut(r,t)\models_A P_i\phi$, so that

$$\exists(a,c'):(a,c')\sim_i Cut(r,t):(a,c')\models_A \phi.$$

By Theorem 10 there exists (r',t') such that $Cut(r',t')=(a,c')$. By Fact 1, $(r',t')\sim_i(r,t)$ and $r'\bowtie r$. Then

$$\exists(r',t'):(r'\bowtie r)\wedge(r',t')\sim_i(r,t):Cut(r',t')\models_A \phi,$$

giving the desired result by the induction hypothesis.

5. C^C: For C^C we return to the view that formulas are functions from sets of global states to sets of global states; again we use subscripts to distinguish the timed-runs semantics from the asynchronous-runs semantics. We

also use the notation that, for any set of consistent cuts Z, Z^* is the set $\{(r,t)|Cut(r,t) \in Z\}$.
We need to show that (r,t) is contained in the greatest fixed point of $\mathcal{M}_T[\![E^C(\phi \wedge X)]\!]$ iff $Cut(r,t)$ is in the greatest fixed point of $\mathcal{M}_A[\![E^C(\phi \wedge X)]\!]$, i.e. B is the later greatest fixed point iff B^* is the former. Repeating the technique for K_i and P_i above, it can be shown that, for any Z,

$$(r,t) \in \mathcal{M}_T[\![E^C(\phi \wedge X)]\!](Z^*) \text{ iff } Cut(r,t) \in \mathcal{M}_A[\![E^C(\phi \wedge X)]\!](Z).$$

It then follows that B^* is a fixed point of $\mathcal{M}_T[\![E^C(\phi \wedge X)]\!]$ iff B is a fixed point of $\mathcal{M}_A[\![E^C(\phi \wedge X)]\!]$, i.e.

$$((r,t) \in B^* \Leftrightarrow (r,t) \in \mathcal{M}_T[\![E^C(\phi \wedge X)]\!](B^*))$$

$$\text{iff}$$

$$(Cut(r,t) \in B \Leftrightarrow Cut(r,t) \in \mathcal{M}_A[\![E^C(\phi \wedge X)]\!](B)).$$

It just remains to be shown that B is the *greatest* fixed point iff B^* is the greatest fixed point also. Suppose that B is but B^* is not. Then there is some set C^*, not a subset of B^*, such that

$$B^* \cup C^* = \mathcal{M}_T[\![E^C(\phi \wedge X)]\!](B^* \cup C^*).$$

However, since (r,t) is contained in $\mathcal{M}_T[\![E^C(\phi \wedge X)]\!](Z^*)$ iff $Cut(r,t)$ is contained in $\mathcal{M}_A[\![E^C(\phi \wedge X)]\!](Z)$, this would be imply that

$$B \cup C = \mathcal{M}_A[\![E^C(\phi \wedge X)]\!](B \cup C).$$

This contradicts the assumption that B is the greatest fixed point. An analogous argument holds in the reverse direction, thus concluding the proof. ∎

4.4 Other Knowledge Forms

There have been other common knowledge variants that are based on replacing simultaneity with weaker temporal notions [HM84]; namely, epsilon common knowledge, C^ϵ, eventual common knowledge, $C^\diamond$, and timestamped common knowledge, C^T. In this section we compare concurrent common knowledge to common knowledge and to each of these variants. As the names indicate, epsilon common knowledge corresponds to agreement within ϵ time units, for some ϵ, eventual common knowledge corresponds to agreement at some future global state of the system, and timestamped common knowledge corresponds to agreement at local states having the same local clock value. The strength of timestamped common knowledge, consequently, depends upon characteristics of the local clocks used.

In the first section we discuss attainment of facts in timed runs and in asynchronous runs; this will be relevant to the comparisons which follow. Then, in the next two sections, we compare C^C to knowledge forms that are unattainable asynchronously. In the final two sections we compare C^C to knowledge forms that are attainable asynchronously.

In Section 4.4.2, we demonstrate that common knowledge is strictly stronger than C^C, by showing that $C \Rightarrow C^C$ and that there are systems where C^C is attained but C is not. In Section 4.4.3, we define C^ϵ and show that it is incomparable to C^C in a strict sense; namely, there are systems and facts ϕ for which $C^\epsilon\phi$ is attainable and $C^C\phi$ is not attainable, and vice versa.

In Section 4.4.4, we demonstrate that C^C and $C^\diamond$ are also incomparable. It should be intuitively clear that eventually agreeing on a fact does not guarantee causal consistency; the converse, however, may not be so clear. It turns out that eventual common knowledge guarantees that a fact ϕ is *known* at points in the run. Concurrent common knowledge, in contrast, only guarantees knowledge of $P_i\phi$ for each i, which is weaker. Consequently, C^C and $C^\diamond$ are indeed incomparable.

In Section 4.4.5, we demonstrate that, in general, timestamped common knowledge, C^T, is incomparable to concurrent common knowledge. We also demonstrate that, in the special case of local clocks being *logical clocks* [Lam77], $C^T\phi$ implies $C^C\phi$ at a consistent cut where local time T is reached by all processes.

4.4.1 Basis for Comparison

In some of the discussions to follow, we will demonstrate systems in which one knowledge form is attained but another is not. Recall, from Section 4.2.1, that we say a fact ϕ is *attained* in run a iff there is a cut c such that $(a, c) \models_A \phi$. Similarly, we say that a fact ϕ is attained in timed run r iff there is a time t such that $(r, t) \models_T \phi$. Recall from Section 4.3 that if $r \in timing(a)$ then r is one of the possible timed runs corresponding to asynchronous run a. When comparing C^C to a knowledge form, say C^X, defined in the timed-runs model, we will consider whether or not attaining $C^X\phi$ in a run r implies attaining $C^C\phi$ in the asynchronous run a such that $r \in timing(a)$, or vice versa. Alternatively, we could carry out these comparisons solely in the timed-runs model, but we use the current method because it is more perspicuous. (If the latter method is used, it is necessary to define what it means to attain $C^C\phi$ in a timed run. Consider any asynchronous run a and timed run r such that $r \in timing(a)$. Furthermore, suppose that fact ϕ is attained in a, i.e. there exists a cut c such that $(a, c) \models_A \phi$. It is not the case that attaining ϕ in a guarantees that, for some t, $(r, t) \models_T \phi$.

Rather, it guarantees that, for some run r' also in $timing(a)$, there exists a t' such that $(r', t') \models_T \phi$. Thus this is the appropriate criterion for attaining C^C in a timed run.)

4.4.2 Common Knowledge

Recall from Chapter 1 that $C\phi$ implies all formulas of the form $E^k\phi$ for any natural number k, where E is defined as follows.

$$E\phi \equiv \bigwedge_{i\in\mathcal{I}} K_i\phi$$

More precisely, C is defined as the greatest fixed point of $E(\phi \wedge X)$, where X is the free variable in the greatest fixed-point operator. We show that this is strictly stronger than C^C below.

Theorem 12. Common knowledge is strictly stronger than concurrent common knowledge, i.e. $C\phi \Rightarrow C^C\phi$ but $C^C\phi \not\Rightarrow C\phi$. Furthermore, there are systems and facts ϕ for which $C^C\phi$ is attained but $C\phi$ is never attained.

Proof: To see that $C\phi \Rightarrow C^C\phi$, recall that $\phi \Rightarrow P_i\phi$. Consequently, $K_i\phi \Rightarrow K_iP_i\phi$ and furthermore, $E\phi \Rightarrow E^C\phi$. Since $C^C\phi$ is the greatest fixed point of $E^C(\phi\wedge X)$ and $C\phi$ is the greatest fixed point of $E(\phi \wedge X)$, the desired result follows.

From [HM84], common knowledge of any fact not initially common knowledge of a system is unattainable in any asynchronous system. We have demonstrated in the current work that concurrent common knowledge of certain facts is attainable in such systems. Thus there are systems and facts ϕ for which $C^C\phi$ is attained and $C\phi$ is not; it follows that $C^C\phi \not\Rightarrow C\phi$. ∎

4.4.3 Epsilon Common Knowledge

Epsilon common knowledge corresponds to agreement within ϵ time units, for some natural number ϵ. Its definition is consequently dependent upon the timed-runs model, since we cannot express "in ϵ time units" in the asynchronous runs model. The formula $\bigcirc^\epsilon\phi$ indicates that ϕ will hold in exactly ϵ time units, as follows.

$$(r, t) \models_T \bigcirc^\epsilon\phi \text{ iff } (r, t+\epsilon) \models_T \phi$$

Epsilon common knowledge is then the greatest fixed point of $\bigcirc^\epsilon E(\phi \wedge X)$, where X is the free variable in the greatest fixed-point operator. $C^\epsilon\phi$ can also be expressed as:

$$C^\epsilon\phi \equiv \forall k : (\bigcirc^\epsilon E)^k\phi$$

We show that C^C and C^ϵ are incomparable in a strict sense: that there are systems and facts ϕ for which $C^C\phi$ is attained and $C^\epsilon\phi$ is not, and vice versa.

Theorem 13. Epsilon common knowledge is incomparable to C^C, i.e. $C^C\phi \not\Rightarrow C^\epsilon\phi$ and $C^\epsilon\phi \not\Rightarrow C^C\phi$. Moreover, there are systems and facts ϕ for which $C^C\phi$ is attained but $C^\epsilon\phi$ is not, and vice versa.

Proof: First, we demonstrate a system $\mathcal{S}_1$ and fact ϕ_1 for which $C^C\phi_1$ is attained, but $C^\epsilon\phi_1$ is not. Let system $\mathcal{S}_1$ contain only two processes, i and j. In this system, the only communication occurs when, at some point, i sends a message m to j along a reliable channel with unbounded transmission time. Let a be any run of this system. Let c be the consistent cut of a consisting of i's local state immediately after sending m and j's local state immediately after receiving m. Clearly c is a cut, since no message is sent after it. It is distinguishable, because its occurrence in each local state is determined by specific events, namely the sending and receiving of m. Let ϕ_1 be the fact "message m has been received." Since ϕ_1 always holds on cut c, from Theorem 7 it follows that $(a, c) \models_A C^C\phi_1$. Therefore $C^C\phi_1$ is attained in any run of the system.

Now consider any r in $timing(a)$. For $C^\epsilon\phi_1$ to be attained, there must be a t such that $(r, t) \models_T C^\epsilon\phi_1$. This implies further that $(r, t+\epsilon) \models_T K_i\phi_1$. However, process i never *knows* ϕ_1, because every local state of i in r is part of a possible real-time state in which message m has not yet been received. Consequently, there is no t for which $(r, t) \models_T C^\epsilon\phi_1$. Therefore $\mathcal{S}_1$ is a system in which $C^C\phi_1$ is attained, but $C^\epsilon\phi_1$ is never attained.

Next, we demonstrate a system $\mathcal{S}_2$ and fact ϕ_2 for which $C^\epsilon\phi_2$ is attained but $C^C\phi_2$ is not. In system $\mathcal{S}_2$, there are three processes i, j, and k which communicate along reliable channels having transmission time bounded by ϵ. Process k can simultaneously send a message to both and i and j. At some point k does send a message m simultaneously to i and j. Also, i and j periodically send messages to each other at random times.

Let ϕ_2 be the fact "message m has been sent by k." $C^\epsilon\phi_2$ holds as soon as m is sent by k. However, $C^C\phi_2$ can never hold, as we shall demonstrate by contradiction. Suppose that $C^C\phi_2$ is attained in a run r of $\mathcal{S}_2$. By Theorem 6, i and j must learn $C^C\phi_2$ along a single consistent cut of r. Let $Learn(i)$ and $Learn(j)$ denote the states of i and j in that cut. $Learn(i)$ must be no earlier than immediately following i's reception of m, and similarly for j; otherwise, processor i (j) could not know $P_i\phi_2$ ($P_j\phi_2$). If $Learn(i)$ does immediately follow i's reception of m, it follows—since $Learn(j)$ is no earlier than j's reception of m and consequently could occur up to ϵ time units later than $Learn(i)$—that a message sent within ϵ time units of $Learn(i)$ could cause the cut to be inconsistent. The same argument is true if $Learn(i)$ occurs anywhere within the first ϵ time units after i receives m. Thus $Learn(i)$ must occur at least ϵ time units after i receives m. A symmetric argument holds for j; $Learn(j)$ must occur at least ϵ time units after j receives m. However, this again results in the situation where the minimum $Learn(i)$ and $Learn(j)$ are ϵ time units apart, as before. We can repeat the argument that $Learn(i)$ and $Learn(j)$ each must occur at least ϵ

time units later, ad infinitum, with the result that $C^C\phi_2$ is never learned. ∎

4.4.4 Eventual Common Knowledge

Eventual common knowledge corresponds to agreement at some future global state of a run. We express it using standard definitions in the timed runs model.

The eventuality operator, $\Diamond$, is defined as follows.

$$(r,t) \models_T \Diamond\phi \text{ iff } \exists\epsilon : \epsilon \geq 0 : (r, t+\epsilon) \models_T \phi$$

Then $E^\diamond$, "everyone eventually knows" is defined as expected.

$$E^\diamond \equiv \bigwedge_{i\in\mathcal{I}} \Diamond K_i\phi$$

Note that, unless facts are stable, $E^\diamond\phi$ does not imply that $E\phi$ ever holds.

Analogously to other common knowledge variants, $C^\diamond\phi$ is defined as the greatest fixed point of $E^\diamond(\phi \wedge X)$. $C^\diamond\phi$ implies, but is strictly stronger than, $(E^\diamond)^k\phi$ for all k:

$$C^\diamond\phi \Rightarrow \forall k : (E^\diamond)^k\phi$$

We prove that this is not only incomparable to $C^C\phi$, but that there are systems and facts for which each is attained but the other is not.

Theorem 14. Eventual common knowledge is incomparable to concurrent common knowledge, i.e. $C^C\phi \not\Rightarrow C^\diamond\phi$ and $C^\diamond\phi \not\Rightarrow C^C\phi$. Moreover, there are systems and facts ϕ for which $C^C\phi$ is attained and $C^\diamond\phi$ is not, and vice versa.

Proof: First, we demonstrate a system and fact ϕ for which $C^C\phi$ is attained, but $C^\diamond\phi$ is not. Consider again system $\mathcal{S}_1$ from the proof of Theorem 13, containing only two processes i and j. Process i sends a single message m to j along a reliable channel with unbounded transmission time. Fact ϕ_1 is "message m has been received." As before, $C^C\phi_1$ is attained in any run of $\mathcal{S}_1$; however, i never *knows* ϕ_1. Since $C^\diamond\phi_1$ implies that eventually $K_i\phi_1$ holds, $C^\diamond\phi_1$ cannot be attained in any run of $\mathcal{S}_1$.

Next, we demonstrate a system and fact ϕ for which $C^\diamond\phi$ is attained, but $C^C\phi$ is not. Consider a system $\mathcal{S}_3$, again with only two processes, i and j. In this system, i sends some number of messages, at least one, to j along reliable, non-FIFO channels. We denote the messages $m_1, m_2, m_3 \ldots$ where message m_k is sent before message m_{k+1}. The sequence numbers are notational and not contained in the messages. Process j never sends a message to i. Let ϕ_3 be "i has sent at least one message to j." Consider any run r of $\mathcal{S}_3$. For any t after i sends the the first message to j, $(r,t) \models_T C^\diamond\phi$.

Let a be the asynchronous run such that $r \in timing(a)$. Suppose that $C^C\phi_3$ is attained in a. By Theorem 6, the states in which i and j learn $C^C\phi_3$ form

a consistent cut of a. Suppose further that j learns $C^C\phi_3$ after receiving one message from j, and that i learns $C^C\phi_3$ after sending message m_k for some k. It is possible—due to the fact that the channels are non-FIFO—that message m_{k+1} is the first to arrive at process j. In this case the cut where $C^C\phi_3$ is learned is inconsistent, contradicting the assumption that j learns $C^C\phi_3$ after receiving one message.

Consider another run a' of $\mathcal{S}_3$ in which only one message is sent from i to j. Since j cannot distinguish a from a' until a second message is received—which could take arbitrarily long since message transmission is unbounded—in run a' process j never learns $C^C\phi_3$. Thus there is a run of $\mathcal{S}_3$ in which $C^C\phi_3$ is never attained, completing the proof. ∎

4.4.5 Timestamped Common Knowledge

Timestamped common knowledge [HM84,NT87] corresponds to agreement at local states having the same local clock value. It is sufficient for our purposes to use the asynchronous-runs model for comparison; we give the appropriate definitions for that model. Let $clock(a, c, i)$ be the value of i's local clock at cut c in run a. Then timestamped knowledge, K_i^T is defined as follows.

$$(a, c) \models_A K_i^T\phi \text{ iff } \forall c' : clock(a, c', i) = T : (a, c') \models_A K_i\phi$$

Note that the satisfaction of $K_i^T\phi$ is dependent only upon the run a, not upon the cut c. The definition of E^T follows the usual pattern.

$$E^T\phi \equiv \bigwedge_{i\in\mathcal{I}} K_i^T\phi$$

Now, timestamped common knowledge is defined as the greatest fixed point of $E^T(\phi \wedge X)$.

The strength of timestamped common knowledge depends upon characteristics of the local clocks used. For example [HM84]:

1. If it is common knowledge that all clocks always show identical times, then at T on any clock, $C^T\phi \Leftrightarrow C\phi$.
2. If ϕ is a stable fact and it is epsilon common knowledge that all clocks are within ϵ time units of each other, then at T on any clock, $C^T\phi \Rightarrow C^\epsilon\phi$.
3. If ϕ is a stable fact and it is eventual common knowledge that all local clocks read T eventually, then at time T on any clock, $C^T\phi \Rightarrow C^\diamond\phi$.

As discussed in Chapter 1, one type of local clock is termed a *logical clock* [Lam77]. Recall that logical clocks have the property that, if event a happens-before event b, then the local clock value at which a occurs is less than the local clock value at which b occurs. This implies that local states having the same clock values are incomparable with respect to happens-before, and consequently form a consistent cut. It might seem that if it is common knowledge that local clocks are logical clocks then $C^T\phi \Rightarrow C^C\phi$. However, there are two problems with this. One issue is that $C^T\phi$ alone does not guarantee that the clock value T is ever reached by any process. Consequently $C^T\phi$ may hold although not all processes know it; in particular, not those whose clocks never reach T. If T is never reached by any process, then $C^T(\mathit{false})$ holds. This is the case for $C^T\phi$ regardless of the local clock properties; it is also the reason that statements like "if it is eventual common knowledge that all local clocks read T eventually" are necessary in the three comparisons above. We will use $\mathit{Reached}(T)$ to denote that clock value T is reached by all processes in a run.

If it is common knowledge that all local clocks are logical clocks, then attainment of $C^T(\mathit{Reached}(T) \wedge \phi)$ does guarantee attainment of $C^C\phi$. Still, it is not the case that $C^T(\mathit{Reached}(T) \wedge \phi) \Rightarrow C^C\phi$; another issue is that $C^T\phi$ holds throughout the run, not just when clocks read T. Only when the clocks read T does $C^T\phi \Rightarrow C^C\phi$. We use the notation $\mathit{At}(T)$ as follows:

$$(a, c) \models_A \mathit{At}(T) \text{ iff } \forall i : \mathit{clock}(a, c, i) = T$$

Thus $\mathit{At}(T)$ implies both that T is reached by all local clocks and that, currently, all clocks have the value T. It is indeed the case the case that

$$C^T(\mathit{At}(T) \wedge \phi) \Rightarrow C^C\phi$$

as we will show in our next theorem.

It is not the case that with logical clocks $C^C\phi \Rightarrow C^T\phi$ for some T. The reasons for this are that (1) logical clocks do not guarantee that every consistent cut contains states having the same local clock value, and (2) as for eventual common knowledge, timestamped common knowledge implies that every process *knows* a fact, which is not guaranteed by concurrent common knowledge.

We formalize these observations regarding timestamped common knowledge with logical clocks in the next two theorems.

Theorem 15. Assume that it is common knowledge that local clock values are logical clocks. Then if $C^T(\mathit{Reached}(T) \wedge \phi)$ is attained by a system then $C^C\phi$ is attained also, and furthermore:

$$C^T(\mathit{At}(T) \wedge \phi) \Rightarrow C^C\phi$$

Proof: Suppose that $C^T(\mathit{Reached}(T) \wedge \phi)$ is attained in run a of a system $\mathcal{S}$. Consider any set of local states in which all clock values read T; there must be at least one set since $\mathit{Reached}(T)$ holds. From the definition of logical clocks, the set must form a consistent cut; call it c. From the definition of $C^T\phi$, $(a, c) \models_A \phi$. Cut c is distinguishable from the occurrence of the local clock value T and the knowledge of $Reached(T)$. Therefore, from Theorem 7, $(a,c) \models_A C^C\phi$. Thus attainment of $C^T(Reached(T) \wedge \phi)$ guarantees attainment of $C^C\phi$. Furthermore, since $C^T(At(T) \wedge \phi)$ implies the conditions on cut c above, $C^T(At(T) \wedge \phi) \Rightarrow C^C(\phi)$. ∎

Theorem 16. If it is common knowledge that local clock values are logical clocks, $C^C\phi \not\Rightarrow C^T\phi$ for any T. Moreover, there are systems and facts ϕ such that $C^C\phi$ is attained but $C^T\phi$ is not attained for any T.

Proof: Consider a system $\mathcal{S}_1'$ as follows. As in system $\mathcal{S}_1$ of Section 4.4.3, there are only two processes i and j. At some point process i sends a single message m to process j. In addition, i has a local clock which begins with the value 1, and increments to 2 immediately after i sends m. Process j has a local clock which begins at 2, and increments to 3 immediately after j receives m. It is simple to verify that the clock values indicated obey the conditions for logical clocks, since m is sent at local time 1 and received at local time 2. Let ϕ_1 be the fact "message m has been received." $C^C\phi_1$ is attained in this system exactly as in Section 4.4.3. However, there is no time T at which ϕ_1 holds when each process reaches T. Furthermore, i never *knows* ϕ_1, which is necessary by the definition of $C^T\phi$. By either argument, $C^T\phi_1$ is not attained for any T in this system. ∎

Finally, we consider general local clocks. Since attaining $C^C\phi$ does not guarantee attainment of $C^T\phi$ in the case of logical clocks, clearly it does not for the general case. Unlike for logical clocks, in the general case $C^T(Reached(T) \wedge \phi)$ does not guarantee attainment of $C^C\phi$.

Theorem 17. For general local clocks, there are systems in which $C^T(Reached(T) \wedge \phi)$ is attained, but $C^C\phi$ is not attained.

Proof: Consider system $\mathcal{S}_3'$ as follows: as in $\mathcal{S}_3$ in the proof of Theorem 14, there are two processors i and j, and i sends a series of messages—at least one—to j. Process j sends no messages to i. Also, in $\mathcal{S}_3'$, each local clock increments each time a message is sent or received. Recall fact ϕ_3, "i has sent at least one message to j." It is straightforward to see that $C^T(Reached(T) \wedge \phi_3)$ is attained in any run of this system, for $T = 1$. However, $C^C\phi_3$ is not attainable in all runs, exactly as in the proof of Theorem 14. ∎

Although $C^T(At(T) \wedge \phi) \Rightarrow C^C\phi$ for systems using logical clocks, this does not preclude the relevance of concurrent common knowledge for such systems. $C^T(At(T) \wedge \phi)$, as shown, is a stronger condition than $C^C\phi$. It implies a useful property for some distributed applications which $C^C\phi$ does not; namely, if two facts ϕ_1 and ϕ_2 become timestamped common knowledge with different timestamps, then all processors learn $C^{T_1}(At(T_1) \wedge \phi_1)$ and $C^{T_2}(At(T_2) \wedge \phi_2)$ in the same order. Known algorithms to achieve C^T with logical clocks [NT87] use two-phase algorithms such as TwoPhase and Tree of Chapter 3. There do not appear to be low-latency protocols such as Flood1 and Flood2 for timestamped common knowledge. Furthermore, concurrent common knowledge does not require processes to keep a local clock value. Thus, for some applications $C^C\phi$ may be a more desirable, though weaker, form of knowledge than $C^T(At(T) \wedge \phi)$. We discuss this issue further in Section 6.3.

Chapter 5

The Role of Inhibition

In this chapter we examine the existence of distinguishable-consistent-cut protocols more closely, as a function of the extent to which they prevent events of the underlying system from occurring.

In all four protocols of Chapter 3, it is necessary to suspend underlying events of the system for an interval. We refer to this suspension as *inhibition.* We desire, however, to separate cases where a protocol only delays events until some number of local actions—sends and internal events—have been performed, as in Flood1 and Flood2, from cases where the protocol delays events while waiting for communication from another process, as in TwoPhase and Tree. We distinguish these as *local* versus *global* inhibition.

In the work of Bougé [BF88], inhibition is termed *freezing* and is defined for a general notion of superimposed processes. However, only the current work and the related work of Critchlow and Taylor [Tay89,Cri89,CT90] have proven fundamental relationships between inhibition and the existence of specific protocols, or distinguished local versus global inhibition.

In the first section we discuss some preliminary work and its relationship to the current work. Next, we formally define inhibition, local inhibition, and global inhibition. In Section 5.3, we characterize the protocols of Chapter 3 according to their inhibitory characteristics. In Sections 5.4 and 5.5, we prove that there is no DCCP for non-FIFO systems using up to local send inhibition and global receive inhibition; we prove some relevant properties about asynchronous systems in the former section before proceeding to the main proof in the latter. In the final section, we summarize further work examining the existence of DCCPs under a variety of system and protocol characteristics, including local and global inhibition.

5.1 Preliminary Work

In preliminary work [Tay89], inhibition and DCCPs are considered within a model that allows an indivisible *receive-send* mechanism. This mechanism allows a process to receive a message and send multiple messages (up to one per neighbor) in response, as a single operation. With the indivisible receive-send model, there is a flooding protocol which is non-inhibitory. That protocol behaves exactly as Flood1 and Flood2, except that the receive-send operation is used rather than suspending system send events or receive events during the sending of protocol messages.

Both existence and message complexities of DCCPs were considered in this model. The following results were proven:

1. There is no non-inhibitory DCCP for non-FIFO systems.
2. Any non-inhibitory DCCP for FIFO systems requires one message per pair of neighboring nodes.

The TwoPhase and Tree protocols apply directly to the receive-send model, and are inhibitory. Recall that these protocols use up to $3nd$ messages total, where d is the diameter of the network. Thus the second fact above proves a precise trade-off between inhibition and message complexity, in systems satisfying the receive-send model. Comparable results for the more general model of our current work are an open question. We discuss this further in our conclusions in Chapter 7.

In the current model, we have given a stronger result regarding the existence of DCCPs for non-FIFO systems; namely, that even local send inhibition and global receive inhibition is insufficient.

5.2 Definitions

The definitions of this section will allow us to characterize nine different categories of protocols with respect to their inhibitory characteristics.

Recall that the function *SysEvents* maps a local state composed of system and protocol events to a local state containing only the system events in their original order. This allows us to define the *disabling* of events by a protocol; an event is disabled in a state if the system events of that state enabled the event in the original system, but the sequence of system and protocol events no longer enables the event.

Definition 30. Let $\mathcal{S}$ be a system, $\mathcal{S} = (\mathcal{I}, \mathcal{C}, Act, En)$, let $\mathcal{P}$ be a protocol, $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, Act', En')$, and let a be a run of $\mathcal{P}(\mathcal{S})$. An event e_i is *disabled* in state s_i of a iff $(SysEvents(s_i), e_i) \in En_i$ and $(s_i, e_i) \notin En'_i$.

If a protocol never disables events then it is termed *non-inhibitory*; otherwise it is termed *inhibitory*.

Definition 31. A protocol $\mathcal{P}$ is *non-inhibitory* if, for all systems $\mathcal{S}$, no event of $\mathcal{S}$ is disabled in any state of any run of $\mathcal{P}(\mathcal{S})$.

A non-inhibitory protocol, then, does not interfere with the running of the underlying system. As noted above, we wish to refine this notion to describe the different degrees of interference exhibited by Flood1 and Flood2 versus TwoPhase and Tree. Thus we further distinguish two kinds of inhibition: *local* versus *global.* A disabled event may be *locally delayed*, in which case the protocol is exhibiting *local inhibition*, or *globally delayed*, in which case the protocol is exhibiting *global inhibition.* We define local delay, intuitively, as follows: an event is locally delayed if it is disabled and, in addition, it is possible that the disabling ceases with no intervention of receive events. An event is *globally delayed* if it is disabled and it is not locally delayed; in other words, communication from another processor is required to re-enable the event.

Definition 32. Suppose event e_i is disabled in state s_i of run a of $\mathcal{P}(\mathcal{S})$. We say that e_i is *locally delayed* if there exists a run a' of $\mathcal{P}(\mathcal{S})$ such that (1) a' contains an extension s_i' of s_i; (2) e_i is not disabled by the protocol in s_i'; and (3) $s_i' - s_i$ contains no receive events.

Given the definitions of local and global delay, the definitions of local and global inhibition are straightforward.

Definition 33. A protocol $\mathcal{P}$ is *locally inhibitory* if it is inhibitory and, for any system $\mathcal{S}$, any event disabled in any run of $\mathcal{P}(\mathcal{S})$ is locally delayed. If $\mathcal{P}$ is inhibitory but does not have this property then it is *globally inhibitory.*

Finally, we also consider whether or not send events or receive events are delayed by an inhibitory protocol.

Definition 34. 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.

Thus, we can define 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. (See Figure 5.4 of Section 5.6.) 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. Consequently we can say that a protocol uses "up to" global send inhibition, meaning that it uses either no delay, local delay, or global delay of send events. This is independent of the type of protocol under consideration.

5.3 Inhibitory Characteristics of Protocols

From the definitions in the previous section we can characterize the inhibition used by the protocols of Chapter 3. In addition to local versus global inhibition we are also concerned with whether send events or receive events are inhibited.

Both of the flooding protocols use local inhibition, since re-enabling of system events occurs only via completion of a series of protocol send events. Flood1 utilizes local *send* inhibition whereas Flood2 utilizes local *receive* inhibition. Both of the two-phase protocols use global inhibition, since a message from the initiator is required to re-enable disabled system events. More specifically, those protocols use global *send* inhibition.

Three properties of the protocols in Chapter 3 are described in Figure 5.1: (1) whether or not they require FIFO channels, (2) the type of inhibition utilized, and (3) whether or not the cut states are consistent with respect to protocol messages. Existence of DCCPs as a function of all of these properties will be discussed in the concluding section of this chapter.

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

Figure 5.1: Varying characteristics of DCCPs.

5.4 Properties of Asynchronous Systems

In this section we present three lemmas concerning the structure of asynchronous systems. These lemmas will be used in the proof that there is no DCCP for non-FIFO systems using up to local send inhibition and global receive inhibition. The first lemma says that any partial run can be extended to a total run.

Lemma 2. If a is a partial run of system S, then there exists at least one total run a' of S that extends it, i.e. $\forall i \,.\, \mathit{Prefix}(a[i], a'[i])$.

Proof: We construct a (possibly infinite) total run a' from a by continually iterating through the set of processes and adding appropriately enabled events. We initialize a' to a. Upon each iteration, for every process i, we arbitrarily pick an event e_i such that either (1) e_i is a send or internal event, and is enabled by $a'[i]$, or (2) e_i is the enabled reception of a message from process j, and the corresponding send event is in $a'[j]$. We term such events *eligible* in $a'[i]$. If there is an eligible event then it is added to a' and we proceed to the next process. Otherwise, we proceed immediately to the next process. Iterations continue indefinitely, unless on some iteration no process has an eligible event.

We must show that this construction results in a total run. Recall that a partial run is a set of local states such that (1) for every receive there is a corresponding send, (2) $\rightarrow$ is a partial order, and (3) for all events there is an enabling state immediately before it in the local state. Clearly each step of the construction results in a new partial run, since the original a' (a) is a partial run and all added events have the necessary enabling local states and corresponding send events. Also, new events cannot cause circularity in $\rightarrow$ since they do not happen-before any event in the partial run.

It is now only necessary to show that the additional constraint on total runs is satisfied, i.e. for each i either (1) $a'[i]$ is infinite, or (2) there is no event e_i enabled by $a'[i]$, unless it is the reception of a message from j and there is no corresponding send event in $a'[j]$. The second condition is equivalent to saying that there are no eligible events in $a'[i]$. The construction above clearly guarantees that any finite history with eligible events will be extended. Thus a' is a total run, completing the proof. ∎

Our next lemma says that, in a non-FIFO system, the reception of a message may occur anywhere in the local history of the receiver where it is enabled as long as the *happens-before* partial order is not violated.

Lemma 3. Let run a of (non-FIFO) system $\mathcal{S}$ contain events e_j on process j and $send(m)$ on process i, where m is from i to j, i.e. there are local states s_i and s_j such that

$$Prefix(s_i \cdot send(m), a[i]) \text{ and } Prefix(s_j \cdot e_j, a[j]).$$

Also, let $s_j \cdot e_j$ not contain $receive(m)$ and let s_j enable $receive(m)$.

If $e_j \not\rightarrow send(m)$ then there exists a run a' also in $\mathcal{S}$ such that

$$Prefix(s_i \cdot send(m), a'[i]) \text{ and } Prefix(s_j \cdot receive(m), a'[j]).$$

(See Figure 5.2. The crossed solid line indicates "not happens-before" and the dashed lines connect the sending and reception of m.)

Proof: The proof proceeds by first showing that there is a partial run a'' of $\mathcal{S}$ where $Prefix(s_i \cdot send(m), a''[i])$ and $a''[i] = s_j$ (without e_j). Then $receive(m)$ is

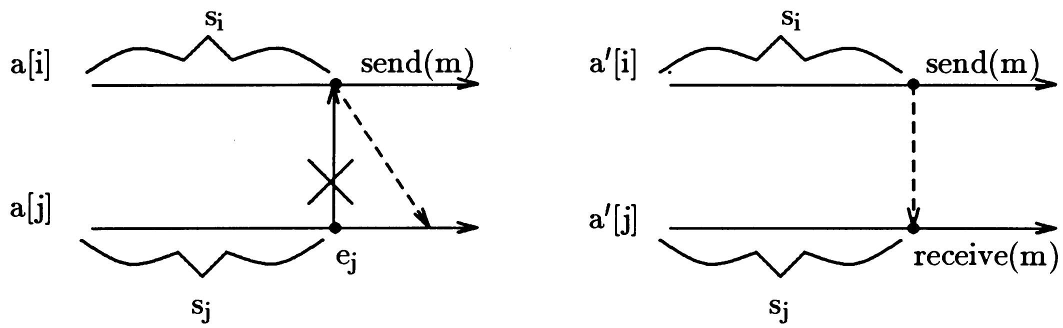

Figure 5.2: Lemma 3.

added to a'' still resulting in a partial run which can be extended to a total run a' by Lemma 2.

We find a partial run of S in a, which satisfies the conditions on a'' above, iteratively. (It is not obvious that such a partial run exists. Perhaps any combination of local states having the required prefixes for i and j will have a message m—potentially involving other processes—which is received but not sent.) This procedure is very similar to the proof of Lemma 1 in Chapter 2. Let $MinSends(s_x, y)$ be the minimum local state of process y in run a which includes the sending of all messages received by x from y in local history s_x. Let $max(s_1, s_2)$, where s_1 and s_2 are local states of a single process in a single run, be the largest sequence of the two (and therefore encompassing the other). Initialize a vector of local states v as follows: $v[j] = s_j$, $v[i] = max(s_i \cdot send(m), MinSends(s_j, i))$, and $v[x] = MinSends(s_j, x)$ for x 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 process p_1 to any other process p_2 whose reception is in v but whose sending is not (if such a message doesn't exist then we are finished). Add events from $a[p_1]$ to $v[p_1]$ until the necessary send is included. As in the proof of Lemma 1, note that, for any inconsistent message m' in the above construction, there is a message chain beginning with m' ending with a message received within the initial state vector. (See Figure 2.2 in Chapter 2.)

In order to meet the conditions of the lemma, we must show that (1) the iterations terminate, and (2) the local state of j is not changed, i.e. upon termination $v[j] = s_j$. Suppose that the iteration above never terminates. Then there must be message chains where two messages m_1 and m_2 are sent by the same process. If $m_1 \rightarrow m_2$ in the chain then also $m_2 \rightarrow m_1$ because there is a local state which includes m_2 but not m_1. This obviously cannot occur in any run. Therefore the iteration terminates.

Suppose that during the iteration state $v[j]$ is changed, due to a message m'. Again, there must be a chain of messages ending with $receive(m^0)$ in the original states and $send(m') \rightarrow receive(m^0)$. There are three cases depending on what process has $receive(m^0)$. (1) If it is a process x, x not i or j, then by the definition of the original set of states v and $MinSends$ there must be a message m_x sent after $receive(m^0)$ from x to j such that $receive(m_x) \rightarrow e_j$ or $receive(m_x) = e_j$; but $e_j \rightarrow send(m')$, $send(m') \rightarrow receive(m^0)$, and $receive(m^0) \rightarrow send(m_x) \rightarrow receive(m_x)$, resulting in an invalid circularity. (2) If it is process j then there is a circularity in $\rightarrow$ as in the proof of termination above. (3) If it is process i then either (a) m^0 arrives before $send(m)$ or (b) m^0 arrives after $send(m)$ but before the sending of a message m_x to j as in the definition of $MinSends$. In case (a),

$$e_j \rightarrow send(m') \rightarrow receive(m^0) \rightarrow send(m)$$

which violates the assumption that $e_j \not\rightarrow send(m)$. Case (b) is analogous to case (1). Hence the final $v[j]$ is s_j.

Let a'' be the final v of the iteration. Then a'' is a partial run of $\mathcal{S}$ such that $a''[j] = s_j$ and $Prefix(s_i \cdot send(m), a''[i])$. Since $receive(m)$ is enabled in s_j, adding event $receive(m)$ to $a''[j]$ still results in a partial run of $\mathcal{S}$. By Lemma 2 this can be extended to a full run a' as the theorem states. ∎

Our final lemma applies to both FIFO and non-FIFO systems. It states that if an event on one process happens-before a non-receive event on another, then it must also happen-before the preceding event in the second local history.

Lemma 4. If run a of (FIFO or non-FIFO) system $\mathcal{S}$ contains events A and B on process i and event C on process j such that (1) $A \mapsto B$, (2) $C \rightarrow B$, and (3) B is not the reception of a message, then $C \rightarrow A$.

Proof: Since C and B are on different processes and B is not a receive, C cannot happen-immediately-before B. Therefore, by the definition of happens-before, there exists events $X_1, ...X_m$ such that

$$C \mapsto X_1 \mapsto ...X_m \mapsto B.$$

Since B is not a receive, only A can happen-immediately-before it. Therefore $X_m = A$ and $C \mapsto X_1 ... \mapsto A$, so $C \rightarrow A$. ∎

5.5 Impossibility Theorem

In this section we prove that there is no non-inhibitory DCCP for non-FIFO systems using up to local send inhibition and global receive inhibition [CT90].

We utilize the following lemma, which is a direct consequence of the definition of local inhibition. It says that, given a consistent cut in which there are disabled events, there is an extension to the cut where all of the disabled events have been re-enabled with no intervening receives.

Lemma 5. Let a be a run of an inhibitory protocol, $(s_1, \ldots, s_N)$ a consistent cut in a, $p_1, \ldots, p_k$ be a subset of processors, and events $e_{p_1}, \ldots, e_{p_k}$ be locally delayed by the protocol in states $s_{p_1}, \ldots, s_{p_k}$, respectively. Then there is a run a' such that $(s_1, \ldots, s_N)$ is a consistent cut in a', and, for $i = 1, \ldots, k$, there are no receive events at processor p_i between s_{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 a_{p_i} containing an extension s'_{p_i} of s_{p_i}, such that $s'_{p_i} - s_{p_i}$ contains no receive events and e_{p_i} is not disabled in s'_{p_i} Let $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$ denote the subsequence of non-receive events between s_{p_i} and s'_{p_i}. In order for run a_{p_i} to exist, there must be pairs (s_{p_i}, e_{i_1}), $(s_{p_i} \cdot e_{i_1}, e_{i_2})$, and so on in the enabling relation for i. Consequently, we can add each of these events to the consistent cut $(s_1, \ldots, s_N)$, resulting in a partial run of the system (note that the fact that the events are non-receives guarantees that conditions (1) and (2) for partial runs are not violated). This generalizes to all other processors. Since any partial run of a system can be extended to a total run of the system, by Lemma 2, the resulting total run satisfies the conditions for a' in the theorem statement. ∎

We can now proceed to a proof of the theorem.

Theorem 18. There is no DCCP for non-FIFO systems using up to local send inhibition and global receive inhibition.

Proof: Given a run a of the protocol, let $s_p \cdot e_p$ be the cut state of p in a for each processor p. Let i be any processor. Let s'_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 s'_i in a. Let j be any processor whose receives are enabled in s'_i; from the manner in which s'_i was chosen, there must be at least one such processor. Let s'_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 s'_j in a. Let s''_j be the last proper prefix of $s'_j \cdot e'_j$ in which receives from i are enabled and e''_j be the event following s''_j. (See Figure 5.3. Note that some of the receive-free intervals may be empty; e.g., it may be that $e_i = e'_i$.)

Let *send*(m_{ij}) be a send event from i to j that is enabled by the subsequence of system events in $s_i \cdot e_i$. Let *send*(m_{ji}) be a send event from j to i that is enabled by the subsequence of system events in $s_j \cdot e_j$. The protocol may inhibit the occurrence of *send*(m_{ij}) in $s_i \cdot e_i$ and of *send*(m_{ji}) in $s_j \cdot e_j$. However, since this inhibition can only be local, by Lemma 5 there is a run a' in which $s_p \cdot e_p$ is a prefix of $a'[p]$ for each p, and no receive events occur between $s_i \cdot e_i$ (respectively $s_j \cdot e_j$) and the state in which *send*(m_{ij}) (respectively *send*(m_{ji})) is re-enabled. We can also assume that *send*(m_{ij}) and *send*(m_{ji}) occur as soon as they are re-enabled. We consider a'.

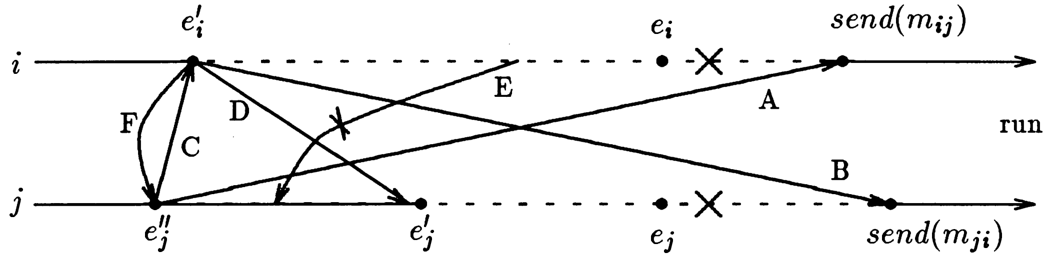

Figure 5.3: Proof of Theorem 18. Dashed lines are intervals in which no receive events occur, arrows represent happens-before, and 'X' marks cut states. (A),(B) follow from Lemma 3. (C),(D) then follow from Lemma 4. (D) and absence of (E) imply (F).

We know that m_{ji} is not received in s_i'; if it were, the cut would be inconsistent. We also know that $receive(m_{ji})$ is enabled by s_i', by definition of s_i'. Suppose that $e_i' \not\rightarrow send(m_{ji})$. Lemma 3 then implies that there is some run a'' of which $s_i' \cdot receive(m_{ji})$ and $s_j \cdot e_j \cdot \ldots \cdot send(m_{ji})$ are prefixes. In this run, neither s_i' nor any prefix of s_i' can be i's cut state: if it were, the same local state would be i's cut state in run a' (by condition (2) of the definition of DCCP), which is not the case. Similarly, $s_j \cdot e_j$ is the cut state of j in a''. In sum, a'' 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 $e_i' \not\rightarrow send(m_{ji})$ in run a' leads to a contradiction. Therefore it must be the case that $e_i' \rightarrow send(m_{ji})$. A symmetric argument establishes that $e_j'' \rightarrow send(m_{ij})$. (See arrows (A) and (B) in Figure 5.3).

Recall that all receive events are disabled between e_j' and e_j, and that none occur between e_j and $send(m_{ji})$. By repeatedly applying Lemma 4, we have that $e_i' \rightarrow send(m_{ji})$ implies that $e_i' \rightarrow e_j'$. Similarly, $e_j'' \rightarrow e_i'$. (See arrows (C) and (D) in Figure 5.3.)

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 5.3). If we can show this, then $e_i' \rightarrow e_j'$ implies $e_i' \rightarrow e_j''$ (arrow (F)) and there will be causal circularity. Suppose there is such a 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 $e_i'' = send(m)$ to processor k. The event $receive(m)$ must occur in k's cut state; otherwise, the remainder of the message sequence will contain an inconsistent message. We can assume that either the event immediately after $send(m_{ij})$ is a system send event $send(m_{ik})$ to k or that sends to k have been disabled by the protocol in the state $s_i \cdot e_i \cdot \ldots \cdot send(m_{ij})$. In the latter case, we can consider (by Lemma 5) a run which is identical to a' up through $send(m_{ij})$, $send(m_{ji})$ and k's cut state, and in which there are no receive events occurring between $send(m_{ij})$ and the state in which some send to k is re-enabled. Assume that the event following this state is $send(m_{ik})$ to k. In either case—$send(m_{ik})$ immediately following $send(m_{ij})$, or after a sequence of non-receive events—necessarily $receive(m) \not\rightarrow send(m_{ik})$. Otherwise, since there are no receive events between $e''_i = send(m)$ and $send(m_{ik})$, $receive(m) \rightarrow send(m_{ik})$ would imply (by Lemma 4) that $receive(m) \rightarrow e''_i = send(m)$, resulting in circular causality. Now, since $receive(m) \not\rightarrow send(m_{ik})$, we can apply Lemma 3 and produce a run in which 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 must be causal circularity in run a'. ∎

5.6 Summary of Impossibility Results

The study of the relationship between inhibition and the existence of DCCPs has been extended by Critchlow and Taylor [Cri89,CT90]. The following protocol and system characteristics are considered: (1) use of local, global, or no inhibition, (2) use of inhibition of send events versus inhibition of receive events, (3) whether or not protocol messages are guaranteed to be consistent with respect to the DCCP cut, and (4) whether or not channels are FIFO. The proof of Theorem 18 is independent of (3), since necessity of system message consistency forces a contradiction.

In [Cri89,CT90], the following impossibility results are proven in addition to Theorem 18.

1. There is no non-inhibitory DCCP for FIFO systems, even if protocol messages are allowed to be inconsistent with respect to the DCCP cut.
2. There is no DCCP for FIFO systems using only local send inhibition if protocol messages are required to be consistent with respect to the DCCP cut.

Figure 5.4 illustrates that these results, along with the protocols of Chapter 3, give a complete characterization of the existence of DCCPs with respect to the conditions outlined above. In the figure, "IMP" designates impossibility, and the IMP† cases are those shown by Theorem 18 and the results above. The remaining impossible cases contain weaker assumptions than the proven results, and therefore follow by direct consequence.

It is perhaps surprising, in the non-FIFO case, that global receive inhibition is insufficient whereas global send inhibition is sufficient. Recall from Section 2.2 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 the use of send inhibition and protocol message consistency. This case is less interesting from our viewpoint, since we are attempting to understand the precise limits on developing DCCPs.

		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	TwoPhase, Tree	TwoPhase, Tree	TwoPhase, Tree	TwoPhase, Tree,Flood1
	Local Receive Inhibition	TwoPhase, Tree	TwoPhase, Tree,Flood2	TwoPhase, Tree	TwoPhase,Tree, Flood1,Flood2
	Global Receive Inhibition	TwoPhase, Tree	TwoPhase, Tree,Flood2	TwoPhase, Tree	TwoPhase,Tree, Flood1,Flood2

Figure 5.4: DCCPs and impossibilities in the inhibition spectrum.

Chapter 6

Applications

In this chapter we discuss applications of our work. The logic presented in Chapter 3, along with the semantics for concurrent knowledge and concurrent common knowledge and their attainability in asynchronous systems, can have the following roles in the development and analysis of distributed algorithms: (1) simplification of solutions and proofs for problems that can be formulated in terms of concurrent knowledge or concurrent common knowledge, (2) characterization of implicit agreement present in certain algorithms, and (3) a tool for reasoning about asynchronous distributed algorithms, particularly with respect to causality. Furthermore, whenever concurrent common knowledge is demonstrated to be sufficient for a particular application, DCCPs provide a solution. When DCCPs are to be used to perform a distributed task, our results on inhibition provide an analysis of necessary system conditions and protocol capabilities.

We will examine several specific situations in which concurrent common knowledge and DCCPs have central roles. In the first section, we prove necessary and sufficient conditions for concurrent actions to take place in distributed systems. In the following two sections, we prove a sufficient condition for broadcasts from a common initiator to arrive in their original order at all locations in the network, without using sequence numbers, and apply this to updating replicated data via DCCPs. In Section 6.4, we show that concurrent common knowledge characterizes the knowledge between two processes attained by a single message transfer along reliable FIFO channels. That result is used in the final section, in which we give a novel analysis of the Chandy-Lamport checkpointing algorithm with regard to both process states and channel states. The expressiveness of our logic has led to short, straightforward proofs for these applications. We assume reliable channels throughout this chapter.

6.1 Concurrent Actions

In the theorem which follows, we use our logic to show a precise necessary and sufficient condition on the performance of *concurrent actions* in distributed systems. Concurrent actions are sets of events which are to be performed concurrently—immediately following a single consistent cut of the system—or not at all. The relationship between concurrent common knowledge and concurrent actions is analogous to that between common knowledge and *simultaneous actions* [MT88] in synchronous systems.

Definition 35. A vector of actions $\overline{\alpha} = \langle \alpha_1, \alpha_2, ..., \alpha_N \rangle$ is a *concurrent action* of system $\mathcal{S}$ iff the following holds. If any element α_i occurs in history $a[i]$ of $\mathcal{S}$, following state s_i, then there is a consistent cut c of a where $(a,c)[i] = s[i]$ and for every j, α_j follows state $(a,c)[j]$.

For example, suppose that each of a set of processes has a local clock, and those local clocks are to be reset concurrently. Then

$$\{\ \alpha_i | \alpha_i \equiv \text{“reset local clock } i \text{ ”}\}$$

is the corresponding vector of actions. We use the operator P_i to give a precise necessary and sufficient condition on the concurrent performance of actions in the next theorem.

In the theorem, if ϕ_i is a *precondition* of an action then i will execute the action immediately following $(a,c)[i]$ iff $(a,c) \models \phi_i$. Reasonable preconditions are determinable from the local state of the process, i.e. if ϕ_i is a precondition and $(a,c) \models \phi_i$ then for any $(a',c') \sim_i (a,c)$, $(a',c') \models \phi_i$. Note that this is equivalent to $\phi_i \Rightarrow K_i\phi_i$. This follows automatically from the positive introspection axiom if ϕ_i is of the form $K_i\psi$. We assume locality of preconditions.

Theorem 19. Let $\overline{\alpha} = \langle \alpha_1, \alpha_2, \ldots, \alpha_N \rangle$ be a vector of actions, for each i let ϕ_i be the precondition of α_i, and let $\overline{\phi} \equiv \bigwedge_{i \in Proc} \phi_i$. Then $\overline{\alpha}$ is a concurrent action in $\mathcal{S}$ iff

$$\bigwedge_{i \in Proc} (\phi_i \Rightarrow K_i P_i \overline{\phi})$$

is valid in $\mathcal{S}$.

Proof: First we show that if the formula is not valid then $\overline{\alpha}$ is not a concurrent action. If the formula is not valid then, for some i and some (a,c), $(a,c) \models \phi_i$ but $\phi_i \not\models K_i P_i \overline{\phi}$. Then, by the definitions of K_i and P_i, for some cut (a',c') that is indistinguishable from (a,c) there is not a cut (a',c'') such that $(a',c'') \sim_i (a',c')$ where ϕ_j holds for all other processes j. By locality $(a',c') \models \phi_i$, so i executes α_i in a'. However, all other processes j do not execute α_j concurrently. Thus $\overline{\alpha}$ is not a concurrent action in $\mathcal{S}$.

Next we show that if the formula is valid then $\overline{\alpha}$ is a concurrent action. Suppose that, for all i and all (a,c), $\phi_i \Rightarrow K_i P_i \overline{\phi}$ and that process i executes α_i following state $(a,c)[i]$. By the definitions of K_i and P_i, there must be a cut (a,c') such that $(a,c')[i] = (a,c)[i]$ and where $\overline{\phi}$ holds. By the definition of precondition all processes j perform action α_j following $(a,c')[j]$. Therefore $\phi_i \Rightarrow K_i P_i \overline{\phi}$ guarantees concurrent performance. ∎

This result is extended by the corollaries below; they give necessary and sufficient conditions on concurrent performance based on concurrent common knowledge.

Corollary 2. Given the conditions of Theorem 19, if $\overline{\alpha}$ is a concurrent action then $\overline{\phi} \Rightarrow C^C \overline{\phi}$ is valid in $\mathcal{S}$.

Proof: From Theorem 19, we have that $\overline{\alpha}$ is a concurrent action implies that $\bigwedge_{i \in Proc}(\phi_i \Rightarrow K_i P_i \overline{\phi})$ is valid. This is turn implies that $\bigwedge_{i \in Proc} \phi_i \Rightarrow \bigwedge_{i \in Proc} K_i P_i \overline{\phi}$ is valid, which is equivalent to $\overline{\phi} \Rightarrow E^C \overline{\phi}$. By the induction axiom, we have that $\overline{\phi} \Rightarrow C^C \overline{\phi}$ is valid. ∎

Corollary 3. Given the conditions of Theorem 19, if $\bigwedge_{i \in Proc}(\phi_i \Rightarrow C^C \overline{\phi})$ is valid in $\mathcal{S}$, then $\overline{\alpha}$ is a concurrent action.

Proof: By the definition of C^C, $C^C(\overline{\phi}) \Rightarrow E^C(\overline{\phi}) \Rightarrow \bigwedge_{i \in Proc} K_i P_i(\overline{\phi})$. Thus if if $\bigwedge_{i \in Proc}(\phi_i \Rightarrow C^C \overline{\phi})$ is valid in $\mathcal{S}$ then $\bigwedge_{i \in Proc}(\phi \Rightarrow K_i P_i(\overline{\phi})$ is valid in $\mathcal{S}$. By Theorem 19, this implies that $\overline{\alpha}$ is a concurrent action. ∎

6.2 Broadcast Ordering

In this section we consider the problem of one process *broadcasting* a sequence of message bodies $\phi_1, \phi_2, \ldots$ to all processes such they arrive everywhere in the order that they are sent.

Definition 36. In an asynchronous run a, a *broadcast of ϕ by process i* is a set of $N-1$ message chains, possibly having some messages in multiple chains, such that (1) all messages contain ϕ as the message body, and (2) each process $j \neq i$ is the recipient of the last message of one of the message chains. Additionally, we say that process i *initiates* the broadcast when it sends the first message in its history that is the first message of one of the message chains.

In general broadcasting a series of message bodies one at a time does not guarantee that they arrive in the same order everywhere, even in FIFO systems, because messages may take different routes of differing transmission speeds. We give a theorem and corollary relating conditions under which (1) message bodies are guaranteed to arrive in the correct order everywhere, and (2) all processes know that they arrive in the correct order everywhere. Let $\rho(\phi)$ denote "message body ϕ has been received by all processes."

Theorem 20. If i knows $P_i(\rho(\phi_k))$ before initiating a broadcast of ϕ_{k+1}, then ϕ_k is guaranteed to arrive before message ϕ_{k+1} at all processes.

Proof: By the definition of P_i, there must be a consistent cut (a, c') where $\rho(\phi_k)$ holds; furthermore, $(a, c')[i]$ precedes the beginning of all of the broadcast message chains. If ϕ_{k+1} arrived at some process j before ϕ_k, hence before $(a, c')[j]$, then cut c' would be inconsistent . ∎

This is a good example of a simple situation where $K_i P_i \psi$ is sufficient to perform an action rather than the stronger traditional knowledge $K_i \psi$. One broadcast does not have to be completed before the next begins to insure the proper ordering, thus alleviating the latency of waiting for acknowledgements.

A result of this theorem is that DCCPs can be used to insure that multiple messages sent to all processes are ordered properly. Although it was only shown previously that $C^C(\alpha)$ is attained with the algorithms invoked with locally-controllable parameter α, the fact "α has been received by all processes" becomes concurrent common knowledge also. This follows from the Attainment Theorem (Theorem 7) because that fact always holds on the locally distinguishable cuts of the algorithms. Thus the algorithms can achieve $C^C \rho(\phi_k)$ which in turn implies $K_i P_i \rho(\phi_k)$.

The theorem above does not imply that the non-initiating processes know that they have received messages in the proper order. This is clearly guaranteed, however, if it is common knowledge that $K_i P_i \rho(\phi_k)$ is a precondition to i initiating the broadcast of ϕ_{k+1}.

Corollary 4. If it is common knowledge that, for all message bodies ϕ_k, $K_i P_i \rho(\phi_k)$ is a precondition to i initiating the broadcast of ϕ_{k+1}, then all processes know that they receive the message sequence in their original order.

6.3 Updates to Replicated Data

In this example, we use the results on broadcast message ordering to develop an algorithm for maintaining consistency of updates to replicated data items. Consider a replicated data item $\overline{x}$, where x_i indicates i's copy of the data. Suppose that process I must perform a sequence of updates to $\overline{x}$ such that these updates occur in the same order at all copies as they do at I.

> **Operation Ordering Problem:** When a process I performs a series of operations, $Op_1, Op_2, \ldots$ that modify its copy x_I of replicated variable $\overline{x}$, ensure that the operations are carried out on each additional copy x_i so that $Op_1(x_i) \rightarrow Op_2(x_i) \rightarrow \ldots$.

Let $\psi_k(y)$ denote "operation Op_k has been performed on variable y by I." The sequence numbers are a notational convenience and are not necessary in the messages sent by processes. In the previous section we observed that DCCPs can insure that a series of broadcasts sent by I arrive at all processes in the order that they are sent. Therefore, a method which uses one of those protocols and in which each process performs operation Op_k immediately upon learning $P_i\rho(\psi_k)$ (or stronger, $C^C\rho(\psi_k)$), solves the Operation Ordering Problem. Using Flood1 as a basis, we obtain the following algorithm.

Protocol 6. Update of Replicated Data $\overline{x}$.

- Initiator I, for each operation Op_k:

  Perform operation $Op_k(x_I)$.

  Send $\langle I, j, \psi_k(x_I)\rangle$ to each neighbor j, before sending any further system messages to j.

- All other processes, i, upon first receiving a message of the form $\langle j, i, \psi_k(x_I)\rangle$:

  Perform operation $Op_k(x_i)$.

  Send $\langle i, k, \phi_k(x_I)\rangle$ to each neighbor $k \neq j$, before sending any further system messages to k.

In contrast, a typical method for ordering operations would be to give each a unique sequence number, then buffer operations until all of those with lower numbers have arrived and been executed. This requires unbounded messages—to accommodate sequence numbers—and buffering of information, neither of which is necessary in Protocol 6.

Note, however, that even concurrent common knowledge of the operations does not handle the case of concurrent updates from multiple initiators, nor does the sequence number method above. If process i makes ψ_1 concurrent common knowledge and process j makes ψ_2 concurrent common knowledge, some other processes, k, may perform Op_1 first whereas others perform Op_2 first. Timestamped common knowledge [HM84,NT87], as discussed in Chapter 4, can guarantee that concurrent broadcasts from different initiators are ordered the same everywhere. However, as noted earlier, known algorithms to achieve C^T appear to require two-phase globally inhibitory protocols, as in TwoPhase and Tree. There do not appear to be lower-latency protocols such as Flood1 and Flood2 for C^T.

Broadcast protocols which achieve each of these two forms of knowledge, C^C and C^T, may be combined to handle replicated data updates efficiently. Such a scheme is used in the ISIS project [BJ87]. ISIS includes two broadcast primitives, CBCAST (*causal broadcast*) and ABCAST (*atomic broadcast*). CBCAST uses a flooding-style protocol as for concurrent common knowledge whereas ABCAST uses a two-phase protocol as for timestamped common knowledge. In ISIS, ABCAST is used to obtain locks for concurrency control of transactions. Once locks are obtained, CBCAST is used to issue operations within each transaction since it is faster and causes no latency. This example illustrates situations in practical systems where two different forms of knowledge are both appropriate characterizations of agreement.

6.4 Single Message Transfer

In this section we show that concurrent common knowledge characterizes the knowledge between two processes attained by a single message transfer along a reliable FIFO channel. A formula subscripted with a set of process identifiers, such as $\{i, j\}$, refers to the subsystem containing only that set of processes.

Theorem 21. A single message m sent along a reliable FIFO channel from i to j achieves

$$C^C_{\{i,j\}}(\text{“}m \text{ has been received”}).$$

Proof: The states immediately following the sending and receiving of m form a locally-distinguishable cut in the $\{i, j\}$ subsystem. Furthermore, "m has been received" holds on this cut. By the Attainment Theorem (Theorem 7), concurrent common knowledge of the fact holds as well. ∎

Theorem 21 enables us to better explain the role of certain messages in the checkpointing example which follows.

6.5 Checkpointing

Our example is an analysis of the agreement implicit in a well-known checkpointing algorithm by Chandy and Lamport [CL85]. The algorithm is used to record global states—including that of channels—for the purpose of detecting global system properties or for rollback recovery. It is designed for asynchronous systems with reliable FIFO channels as in our model.

The algorithm is a variant of that described in [CL85] and is very similar to Flood1. The algorithm works by using rules for sending and receiving special types of messages called *markers*. Initialization occurs by one process following the marker sending rule.

Marker sending rule for a process i: Process i records its state, then sends one marker to *each* neighbor j. While recording its state and sending the markers, i does not send any system messages.

Marker receiving rule for a process j: Upon receiving a marker from i, if j has not recorded its state, then j follows the marker sending rule, then records the state of channel (i,j) as the empty sequence. Otherwise, j records the state of channel (i,j) as the sequence of messages received from i after j's state was recorded and before j received the marker from i.

A perhaps subtle but important difference between this algorithm and Flood1 is that a marker is sent along every channel, including a marker back along the channel where a process received its first marker. These extra messages give information about the state of channels during the checkpoint. The next theorem summarizes the agreement attained in the algorithm regarding both the process states and the channel states.

The theorem statement makes use of some additional notation. Given any consistent global state, $current_i$ will denote i's local state. $Saved_i$ denotes the set of states checkpointed by process i. $Learn_i(\psi)$ indicates the local state in which i learns ψ. Finally, we extend $\rightarrow$ to include states in the obvious way.

Theorem 22. In any run of the Chandy-Lamport protocol:

1. $C^C(\phi_{states})$ is attained, where

$$\phi_{states} \equiv \forall i : current_i \in Saved_i$$

2. for each channel (i,j), $C^C_{\{i,j\}}(\phi_{channels})$ is attained, where

$$\phi_{channels} \equiv \forall m :$$
$$send(m) \rightarrow Learn_i P_i C^C(\phi_{states}) \Rightarrow receive(m) \rightarrow current_j$$

   where m is a message from i to j.

Informally, the first statement above says that there is a consistent cut of the system in which all processes have concurrent common knowledge that all processes are taking a checkpoint. We will not give a proof as this is a simple variation of Flood1.

The second statement says that—within each (i,j) subsystem—it becomes concurrent common knowledge that all messages sent prior to i recording its state have been received by j. Hence when this is attained, the messages in the (i,j) channel during the checkpoint are exactly those that have been received since j recorded its state. This follows from the fact that all messages from i to j are in one of three states at the checkpoint:

1. Received before the checkpoint. These messages are part of j's local state at the time of the checkpoint.
2. In the channel during the checkpoint. All messages not in (1) received prior to $Learn_j P_j(\phi_{channels})$ must be in this set.

3. Sent after the checkpoint. These messages are not received by j until after $Learn_j P_j(\phi_{channels})$.

Proof of Theorem 22, part 2: From Theorem 21, the reception of the marker by j becomes concurrent common knowledge between i and j in the subsystem cut formed by the states immediately following the sending and receiving of the marker. Since the marker is sent after $Learn_i P_i(C^C(\phi_{states}))$ and channels are FIFO, messages sent prior to $Learn_i P_i(C^C(\phi_{states}))$ must be received before the marker. Therefore, at the subsystem cut it becomes concurrent common knowledge of i and j that all messages sent prior to $Learn_i P_i(C^C(\phi_{states}))$ have been received. ∎

Any problem that requires only the detection of some property of a consistent global state can be solved using the Chandy-Lamport algorithm, or other DCCP variants. Some examples of this type of problem are termination detection [Fra80], deadlock detection [BT87], and rollback recovery [KT87]. Hence concurrent common knowledge gives a concise formal analysis of such problems. Furthermore, our results on inhibition apply whenever one is using DCCPs to detect global state properties in asynchronous systems.

Chapter 7

Conclusions

We have made two significant contributions to understanding the structure and capabilities of asynchronous systems: we have given a new logic and semantics for reasoning about agreement and causality in asynchronous systems, and we have proven strong relationships between inhibition and the achievement of causal consistency in a common class of asynchronous distributed protocols.

7.1 Concurrent Common Knowledge

Concurrent common knowledge is defined in terms of the causality relation between events, which is an appropriate notion of order when one is discussing asynchronous systems [Lam77]. We use a modal logic and semantics that are designed specifically to capture only the causal structure relevant to such systems.

We have given a general condition under which concurrent common knowledge is attained in distributed systems, and shown that distinguishable-consistent-cut protocols can be used to attain concurrent common knowledge of certain facts. We have given four simple and efficient DCCPs, thus demonstrating that concurrent common knowledge is efficiently attainable in asynchronous systems. We have also presented several distributed applications in which the attainment of concurrent common knowledge is highly relevant. It is the attainability and broad applicability of concurrent common knowledge that makes it an important concept for asynchronous distributed computing.

One of the contributions of our work is that we have given a knowledge-theoretic definition that applies whenever one needs to reason about the global states of asynchronous systems. Thus we have described the form of knowledge a protocol designer should try to attain when developing a protocol to reach agreement about some property of a consistent global state. We have also used our logic to prove necessary and sufficient conditions for performing concurrent actions in asynchronous distributed systems.

There have been other proposals for weakened forms of common knowledge that are also attainable asynchronously, namely eventual common knowledge and timestamped common knowledge [HM84]. They use various temporal modalities in order to weaken the original definition of common knowledge. Concurrent common knowledge is strictly weaker than common knowledge but is, in general, incomparable with the other forms above. In the case of timestamped common knowledge, if the clocks used in its definition are logical clocks as in [NT87], and the timestamp of interest is guaranteed to be reached by all processes, then attainment of timestamped common knowledge guarantees attainment of concurrent common knowledge. However, in practice timestamped common knowledge seems to require inhibition and longer latency in cases where concurrent common knowledge does not. It also requires the existence of a local process clock value whereas concurrent common knowledge does not. Thus achieving concurrent common knowledge may be more practical when it is sufficient for a particular application.

In the future, we hope to use our logic to understand better the communication requirements of a wider variety of asynchronous distributed protocols, and to aid in developing new and improved protocols. We also hope to extend the usefulness of our logic by addressing the issue of faulty environments.

The failure of processes and channels affects both the definition and attainability of concurrent common knowledge. We have made some preliminary observations regarding this issue. If process failures are considered, it is probably desirable to define concurrent common knowledge relative to the set of non-faulty processes using indexical sets as done by Moses and Tuttle [MT88]. It appears that with halting-style process failures, or crash failures, a modified notion of concurrent common knowledge amongst the non-faulty processes is attainable. In the case of channel failures by message loss, attainability appears to depend on the ability to save and retransmit entire message histories. Finally, if we consider omissions failures by processes, in which a faulty process may occasionally fail to send or receive messages, attainability appears to depend on whether or not the failed (but not halted) processes are also required to attain concurrent common knowledge. We plan to formalize these observations in the future, thus extending the usefulness of concurrent common knowledge to much broader classes of protocols.

7.2 Inhibition and DCCPs

We have formalized the notion of inhibitory protocols, those that cause some delay of underlying system events, for asynchronous distributed systems. In the work of Bougé [BF88], inhibition is termed *freezing* and is defined for a general notion of superimposed processes. However, only the current work and related work of Critchlow and Taylor [Tay89,Cri89,CT90] have proven fundamental relationships between inhibition and the existence of specific protocols, or distinguished local versus global inhibition.

We prove that there is no DCCP for non-FIFO systems without the global inhibition of send events. Extensions to this work [Cri89,CT90] have resulted in a complete analysis on the existence of DCCPs as a function of the following protocol and system characteristics: local versus global inhibition, inhibition of send events versus receive events, the FIFO or non-FIFO nature of channels, and whether or not protocol messages are allowed to be inconsistent with respect to the DCCP cut. In particular, it has been also shown that local inhibition is necessary and sufficient to develop a DCCP for FIFO systems.

Related work by Awerbuch [Awe85] examines message versus time complexity trade-offs in a class of protocols, called *synchronizers*, which resemble DCCPs. A message-complexity lower bound for synchronizers is given as a decreasing function of time complexity. However, Awerbuch does not consider inhibition directly, although it is a potential source for increasing time complexities of protocols. Also, synchronizers and DCCPs differ in their causal constraints.

There are many areas of potential future research with regard to inhibition and DCCPs. 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. Our model does not allow this method, because of the separation of protocol and system events. There are non-FIFO consistent-cut protocols which use piggybacking [LY87,Mat89] and are, informally, non-inhibitory. However, either they do not have the distinguishability characteristic of DCCPs, or they require the ability to "peek" at message contents before actually receiving them. Another approach is to simulate FIFO channels by including the entire message history with every message; however, this requires unbounded message size, and issues of inhibition and distinguishability have yet to be formalized for such a model.

Another important issue is the relative complexities of DCCPs. It appears that results of [Tay89] can be extended to show that any DCCP for FIFO systems using only local inhibition requires one message per pair of neighboring nodes. Since DCCPs using global inhibition can use as little as $3(N-1)$ messages, as in the Tree protocol, this would illustrate a trade-off between global inhibition and message complexity. A lower bound for protocols using global inhibition is also under investigation.

Finally, it is our expectation that we can prove relationships between inhibition and the attainability and message complexities of protocol classes other than DCCPs. Our definitions of inhibition and of local and global inhibition are generally applicable to distributed protocols; they are independent of asynchrony and DCCPs. Thus we feel that they have a strong potential for the analysis of wider varieties of distributed protocols.

Bibliography

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J.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.

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

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

[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.

[CM86] M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

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

[CT90] Carol Critchlow and Kim Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report 90-1101, Cornell University Department of Computer Science, February 1990. To appear in *A.C.M. Symposium on Principles of Distributed Computing*, August 1990.

[DM86] C. Dwork and Y. Moses. Knowledge and common knowledge in a byzantine environment: The case of crash failures. In J. Halpern, editor, *Proceedings of the Conference on Theoretical Aspects of Reasoning About*

Knowledge, pages 149–170. Morgan Kaufmann, 1986. To appear in *Information and Computation*, 1990.

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

[Gra78] Jim Gray. Notes on database operating systems. In *Lecture Notes in Computer Science 60*. Springer-Verlag, Berlin, 1978.

[GT89] A. Gopal and S. Toueg, January 1989. Personal communication.

[HF88] J. Y. Halpern and R. Fagin. Modelling knowledge and action in distributed systems. 1988. To appear in Distributed Computing.

[Hin62] J. Hintikka. *Knowledge and Belief*. Cornell University Press, 1962.

[HM84] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the Third A.C.M. Symposium on Principles of Distributed Computing*, pages 50–61, 1984. A revised version appears as IBM Research Report RJ 4421, Aug. 1987; to appear in J.A.C.M.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[Lam77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the A.C.M.*, 21(7):558–565, 1977.

[Lam85] L. Lamport. Paradigms for distributed computing. *Methods and tools for specification, an advanced course* (M. Paul and H.J. Siegert, eds.), Lecture Notes in Computer Science 190, pages 19–30,454–468, 1985.

[Leh84] D. Lehmann. Knowledge, common knowledge and related puzzles. In *Proceedings of the third A.C.M. Conference on Principles of Distributed Computing*, pages 62–67, 1984.

[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.

[MT88] Y. Moses and M. Tuttle. Programming simultaneous aactions using common knowledge. *Algorithmica*, 3:121–169, 1988. A preliminary version appeared in the 27th IEEE Symp. on Foundations of Computer Science, 1986.

[NT87] Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *Proceedings of the Sixth A.C.M. Symposium on Principles of Distributed Computing*, pages 281–293, 1987. To appear in J.A.C.M.

[PR85] R. Parikh and R. Ramanujam. Distributed processing and the logic of knowledge. In *Proceedings of the Brooklyn College Workshop on Logics of Programs*, pages 256–268, 1985.

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

[Tar55] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tay89] K. Taylor. The role of inhibition in asynchronous consistent-cut protocols. *Proceedings of the Third International Workshop on Distributed Algorithms,* Lecture Notes in Computer Science 392, pages 280–291, 1989.

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

[YC79] Y. Yemini and D. Cohen. Some issues in distributed process communication. In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 199–203, 1979.