The Inhibition Spectrum and the Achievement of Causal Consistency (Extended Abstract)

Carol Critchlow
Center for Applied Mathematics

Kim Taylor *
Dept. of Computer Science

Cornell University
Ithaca, New York 14850

Abstract

We consider the problem of distinguishing causally-consistent global states in asynchronous distributed systems. Such states are fundamental to asynchronous systems, because they correspond to possible simultaneous global states; their detection arises in a variety of distributed applications, including global checkpointing, deadlock detection, termination detection, and broadcasting. We consider a spectrum of protocol capabilities based on the type of *inhibition* that occurs, i.e. the extent to which the protocol delays events of the underlying system. For the first time we distinguish *local* versus *global* inhibition and prove fundamental relationships between these concepts and determining causally-consistent states. In local inhibition, processors only delay events until they have performed some number of local actions; in global inhibition, they delay events while waiting for some communication from other processors. Based on a variety of system and protocol characteristics, including the ability to locally or globally inhibit particular types of events, we give several new impossibility results and examine some existing protocols. We are then able to present a thirty-six-case summary of protocols and impossibility results for the determination of causally-consistent states as a function of those characteristics. In particular, we demonstrate that local inhibition is necessary and sufficient to solve this problem for general FIFO systems, while global send inhibition is necessary for general non-FIFO systems.

*Supported by an AT&T Ph.D. Scholarship.

1 Introduction

We consider the problem of distinguishing causally-consistent global states in asynchronous distributed systems. Lamport [13] introduces *causality*, a means of providing temporal structure to asynchronous systems. Causality may be used to define a *consistent global state* [16,5] of an asynchronous system, sometimes referred to as a *consistent cut*. Consistent cuts are fundamental to asynchronous systems, as they correspond to possible simultaneous global states. We address the class of consistent-cut protocols, or *CCPs*, protocols in which such causally-consistent states are determined. CCPs arise in numerous distributed applications such as system checkpointing [16,5,11], deadlock detection [4], distributed termination [9], and broadcasting [2].

Our spectrum is based on the *inhibitory* capabilities of protocols. Inhibition in asynchronous distributed systems is formally defined in [17]. In that work, inhibition refers to protocols delaying actions of the underlying system for an interval. For example, a two-phase CCP is given which de-

lays the sending of messages between phases. In [3], this concept is termed *freezing* and is defined for a general notion of superimposed processes. However, only the related works of [17,8] have proven fundamental relationships between inhibition and determining consistent global states—this paper extends those works significantly—and no previous work has made the important distinction between *local* and *global* inhibition.

In [17] it is shown that there is no non-inhibitory CCP for non-FIFO systems, and a non-inhibitory CCP is given for FIFO systems. However, the non-inhibitory CCP uses an indivisible local operation for receiving a message and sending multiple messages in response. This is certainly "inhibitory" in a sense. In [8], it is shown that indeed there is no non-inhibitory CCP for FIFO systems without this atomic receive-send mechanism. We refine the previous works by separating cases where a protocol only delays events until some number of local actions—sends and internal events—have been performed from cases where it delays events while a processor waits for communication from another processor. We distinguish these as local versus global inhibition. Local inhibition can be used to form indivisible local operations which perform multiple tasks, such as the atomic receive-send referred to above. The distinction of local and global inhibition has enabled us to examine the limitations on developing CCPs more precisely than is accomplished in [17,8].

In addition to the possibility of local and global inhibition, we also consider whether (1) sends, receives, or both can be delayed, (2) the system is FIFO or non-FIFO, and (3) protocol messages are allowed to be inconsistent with respect to the cut designated by the protocol. Regarding (3), previous works on consistent global states have assumed one case or the other implicitly; however, this assumption affects the success of some protocols and the existence of protocols under certain conditions. We give the following results for the first time.

1. There is no CCP for non-FIFO systems with local send inhibition and global receive inhibition, even if protocol messages are allowed to be inconsistent.
2. There is no CCP for FIFO systems with no inhibition, even with inconsistent protocol messages.
3. There is no CCP for FIFO systems with local send inhibition but no receive inhibition, if protocol messages must be consistent.

For (1) we prove that, in order to prevent message inconsistency, some pair of neighboring processors must have causal circularity between the last states preceding the cut in which those processors are willing to receive messages from each other. We prove (3) in the full paper; (2) is proven in [8]. The latter two proofs are very similar; in both, we demonstrate the necessity of certain "essential" messages, sent between processors before reaching the cut, and then prove the existence of a run in which some process is forced to reach its cut state before sending all of its essential messages.

We also give three CCPs and discuss their inhibitory nature, along with conditions under which they are successful. All of these results are organized into a thirty-six-case summary of protocols and impossibility results as a function of system assumptions and protocol capabilities. (See Figure 2 of Section 4.)

2 System Model

We model asynchronous distributed systems; more specifically, sets of autonomous processors which can communicate by sending and receiving messages along bi-directional channels. By asynchronous, we mean that (1) there is no global clock, (2) there is finite but unbounded delay in the transmission of messages, and (3) processors may proceed at different rates. The network is assumed to be connected though not necessarily completely connected. Systems are assumed to be reliable: no processor can fail, no message can be lost or altered in transmission, and all messages received have been sent.

Our model is similar to other models of asynchronous systems [6,15]. The system behavior is represented by a set of possible runs; each run

is composed of sequences of events corresponding to the local actions of each process. Unlike [6,15], we explicitly model the *enabling* of events by preceding event sequences [18]. We also explicitly separate protocol and system events. These two differences allow us to define inhibition by comparing how an underlying system enables events versus how the system plus the protocol enables events. Our model is a generalization of [17], which contains an indivisible operation for receiving a message and sending multiple responses. In our model, the only indivisible operations are the sending or receiving of single messages, in addition to internal events.

Each system consists of a connected network of N processors joined by bi-directional channels. We let $\mathcal{I} = \{1, 2, \ldots, N\}$ denote the set of process identifiers. We let $\mathcal{C}$ denote the set of bi-directional channels, consisting of unordered pairs of distinct elements from $\mathcal{I}$. If the unordered pair (i, j) is in $\mathcal{C}$ then processor i can send messages to processor j, and vice-versa; we then term i and j *neighbors.*

With each processor i is associated a set E_i of *events.* A sequence of events from E_i is called a *local history* of processor i, and any such finite sequence is a *local state* of i. We let the set $States(E_i)$ denote the set of all possible local states consisting of events from E_i. Events are of three kinds: send events, in which one message is sent to a neighbor; receive events, in which one message is received from a neighbor; and internal events.

Also associated with processor i is an *enabling relation* M_i on local states and events:

$$M_i \subseteq States(E_i) \times E_i$$

If the pair (l_i, e_i) is in M_i, where l_i is a local state of processor i and $e_i \in E_i$, then we say l_i *enables* e_i. The enabling relation describes what events *could* occur in a given local state of processor i; there may be several events enabled by a single local state. We require the following: for each processor i and each local state l_i of i, if $(l_i, receive(m)) \in M_i$, where m is a message from processor j, then $(l_i, receive(m')) \in M_i$ 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 at another, but may not selectively enable receives on a single channel based on message contents. Consequences of the alternative assumption are discussed at the end of Section 4.

Using the elements described thus far, we can now formally define a system. A *system* $\mathcal{S}$ is a quadruple $(\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$, where $\mathcal{I} = \{1, \ldots, N\}$ is a set of process identifiers, $\mathcal{C}$ is a set of unordered pairs of distinct processors from $\mathcal{I}$, $\mathcal{E} = \{E_1, E_2, \ldots, E_N\}$ is an N-vector of sets of events, and $\mathcal{M} = \{M_1, M_2, \ldots, M_N\}$ is an N-vector of enabling relations, such that (1) for each process i, $M_i \subseteq States(E_i) \times E_i$, (2) for each message from process i to process j in $\mathcal{E}$, (i, j) is in $\mathcal{C}$, and (3) for any two processes i and j, there is a path $(i, i_1), (i_1, i_2,), \ldots, (i_k, j)$ in $\mathcal{C}$ (i.e. the network is connected).

We will associate a set of possible runs, or executions, with each system. First, we introduce a means of providing temporal structure to sets of local histories, as in [13]. This temporal structure expresses the fact that certain events precede other events and could, therefore, have a causal effect on them. Hence this is referred to as *potential causality*, or simply causality. Given an N-vector $r = (r_1, \ldots, r_N)$, where r_i is a local history of processor i for each i, and two events e and e' in r, then e *happens-before* e' (written $e \rightarrow e'$) if either (1) e and e' are both events in r_i for some i, and e occurs before e' in r_i; (2) e is $send(m)$ and e' is $receive(m)$ for some message m; or (3) (e, e') is in the transitive closure of pairs of events satisfying conditions (1) and (2) above, i.e. there is an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.

We also define a completeness condition on the local histories of each execution; more specifically, they cannot contain *forever-enabled* events. An event is forever-enabled in a local history if it is enabled by some local state l of that local history, it is not in the history, and it is enabled by all local states l' for which l is a prefix.

We can now associate a set of *total runs* with each system, corresponding to the possible executions of that system. A vector r of local histories is a *total run* of a system if five conditions hold: (1) for all prefixes $l_i \cdot e_i$ of r_i, l_i enables e_i

$(1 \leq i \leq N)$; (2) there is no *receive*(m) for which there is not a corresponding *send*(m); (3) the reflexive closure of happens-before is a partial order on the set of events included in r, i.e. there are no two events e and e' such that $e \rightarrow e'$ and $e' \rightarrow e$ in r; (4) no send or internal event of i is forever-enabled in r_i; and (5) no receive event of processor i for which there is a corresponding send event in r is forever-enabled in r_i. If the first three conditions hold of a vector $r' = (r'_1, \ldots, r'_N)$ of local states, then r' is called a *partial run.* If r is a total run such that r'_i is a prefix of local history r_i for all i, then r' is also called a *consistent cut* of r.

We define a protocol in an additive fashion, so as to separate the behavior of the protocol from the behavior of an underlying system. Given a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$, a protocol $\mathcal{P}$ will produce a new system $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}')$. The events of $\mathcal{E}$ are called *system events* and the events of $\mathcal{E}' - \mathcal{E}$ are called *protocol events.* Given a sequence of protocol and system events, the function *SysEvents* returns the projection of the sequence onto the set of system events; this may be applied to local states or local histories.

Formally, a *protocol* $\mathcal{P}$ is a function which maps a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$ to a new system $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}')$ such that (1) $\mathcal{E} \subseteq \mathcal{E}'$ (2) $\mathcal{M}' = (M'_1, ..., M'_N)$, where $M'_i \subseteq States(E'_i) \times E'_i$, and (3) for all total runs $r' = (r'_1, ..., r'_N)$ of $\mathcal{P}(\mathcal{S})$, where r'_i is the local history of process i, run $r = (SysEvents(r'_1), ..., SysEvents(r'_N))$ is a total run of the original system $\mathcal{S}$. We refer to runs of this modified system as *runs of the protocol* with respect to the original system. In condition (3), we require that the projection of a protocol run onto the set of system events be some run of the original system. This implies that some valid behavior of the original system results.

We investigate the existence of *consistent-cut protocols* for our systems, protocols that distinguish a set of local states in every run that form a consistent cut. These differ from *snapshot* [5,11] protocols in that we are not concerned with recording the states of channels. Of course, impossibility results for consistent-cut protocols directly apply to protocols, such as snapshots, that perform other tasks in addition to determining consistent cuts. For simplicity, we consider protocols which distinguish a single consistent cut in each run. Our results generalize in a straightforward manner to protocols distinguishing multiple non-intersecting consistent cuts.

Definition 1. A *consistent-cut protocol (CCP)* is a protocol $\mathcal{P}$ which, for every system $\mathcal{S}$ and every protocol run r in $\mathcal{P}(\mathcal{S})$, will create a set of local states $cut(r) = (l_1, \ldots, l_N)$ called *cut states*, such that (1) $(l_1, \ldots, l_N)$ is a consistent cut; (2) in every run $r' = (r'_1 \ldots, r'_N)$ in which l_i is a prefix of r'_i, l_i will be the ith component of $cut(r')$, i.e. if l_i is i's cut state in run r, then it will be i's cut state in all runs in which it occurs; and (3) if r is a run of $\mathcal{S}$ and c is a consistent cut of r, then there is a run r' of $\mathcal{P}(\mathcal{S})$ which extends c, i.e. for each i, c_i is a prefix of r'_i.

The second condition implies that the cut state of each process is distinguishable to that process. In other words, each processor i "knows" [15] that any messages sent subsequently to j will not be received in j's cut state, regardless of the sequence of steps that j may have taken. The third condition implies that a CCP cut may occur anywhere in the run of the underlying system; this eliminates CCPs which, for example, only distinguish a cut at the beginning of a run. We sometimes consider CCPs in which $cut(r)$ may contain inconsistent *protocol* messages; this implies that condition (1) is changed to require that $(SysEvents(l_1), \ldots, SysEvents(l_N))$ be consistent.

Because of the separation of system and protocol events, our model does not allow the merging of system and protocol messages. This type of merging occurs in *piggybacking* techniques, in which protocol information is added to the contents of system messages. Our results are dependent upon this restriction. There are protocols, utilizing piggybacking, that determine consistent cuts of non-FIFO systems and are "non-inhibitory" in a sense [10,12,14], although they typically do not have the distinguishability characteristic (condition (2) of the CCP definition). We address this further in Section 6. Our results show that, without piggybacking or some other

mechanism stronger than those of our model, a CCP for non-FIFO systems cannot be attained, even with a high degree of inhibition.

3 The Inhibition Spectrum

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

First, we define inhibition essentially as in [17]. We take the intermediate approach of first defining the notion of *disabling* an event.

Definition 2. Let $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$ be a system, $\mathcal{P}$ be a protocol with $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}')$, and r be a run of $\mathcal{P}(\mathcal{S})$. An event e_i is *disabled* in state l_i of r if the subsequence of system events in l_i enables e_i in the original system, but l_i does not enable e_i:

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

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

Definition 3. A protocol is *non-inhibitory* if no event is disabled in any run of the protocol.

A non-inhibitory protocol, then, does not interfere with the running of the underlying system. Any protocol which does disable system events is *inhibitory*.

Recall that we wish to separate cases where a protocol delays events only until some number of local actions—send or internal events—have been performed from cases where it delays events while waiting for communication from another processor. Hence we further distinguish two types of inhibition, *local* and *global*, for the first time. We use the terminology that an event may be locally or globally *delayed*, whereas in doing so a protocol exhibits local or global *inhibition*.

Definition 4. Suppose event e_p is disabled by a protocol in state l_p of run r. We say e_p is *locally delayed* if there exists a run r' such that (1) r' contains an extension l_p' of l_p; (2) e_p is not disabled by the protocol in l_p'; and (3) $l_p' - l_p$ contains no receive events.

Thus, the re-enabling of a locally delayed event does not depend upon communication from another processor. Local delay has the following consequence: if events have been locally delayed in the states of a partial run, then in some extending run, each event is re-enabled with no intervening receives. This is formalized by the following lemma; its proof is straightforward from the definition of local delay and is contained in the full paper.

Lemma 1. Let r be a protocol run, $(l_1, \ldots, l_N)$ be a consistent cut of r, $p_1, \ldots, p_k$ be a subset of processors, and events $e_{p_1}, \ldots, e_{p_k}$ be locally delayed by the protocol in states $l_{p_1}, \ldots, l_{p_k}$, respectively. Then there is a run r' such that $(l_1, \ldots, l_N)$ is a consistent cut of r', and, for $i = 1, \ldots, k$, there are no receive events between l_{p_i} and a state in which e_{p_i} is no longer disabled.

Given the definition of local delay, locally and globally inhibitory protocols are defined in a straightforward manner.

Definition 5. A protocol is *locally inhibitory* if any event disabled in any run of the protocol is delayed locally. An inhibitory protocol that does not have this property is *globally inhibitory*.

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

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

Definition 6. 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 2 of Section 4.) For example, a protocol exhibits local receive inhibition and global send inhibition

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

Figure 1: Protocol characteristics.

if it delays both receive and send events, and no delayed receive event requires communication to be re-enabled.

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

4 Protocols

In this section we give three CCPs and discuss their inhibitory characteristics. The first two are *flooding* algorithms, in which messages are sent along every channel in the system [7,5]. *Flood1* is essentially the Chandy and Lamport checkpointing algorithm [5]. *Flood2* is similar to that of [17], with the indivisible receive-send mechanism replaced by local receive inhibition. The third algorithm, *Tree*, is a two-phase spanning tree protocol from [17]; it is similar to protocols in [9,1]. Properties of these CCPs are summarized in Figure 1, namely: (1) whether they work for non-FIFO channels, or only FIFO, (2) the type of inhibition used, and (3) the consistency of the resulting cut with respect to protocol messages. We leave a discussion of message complexity to Section 6.

In both Flood1 and Flood2, messages are sent along every channel in the system beginning with a distinguished initiator I. A consistent cut 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 inhibited during the interval in which protocol messages are sent. In Flood1, in which the cut is reached at the beginning of that interval, system messages sent to a neighbor prior to the protocol message could be inconsistent and are therefore inhibited. In Flood2, the cut is reached at the end of the interval. Consequently, it is the reception of system messages which could cause inconsistency and must be inhibited. In both protocols, the inhibition is only local because a sequence of send events ends the disabling, with no necessary communication from another processor.

Each $(*)$ below indicates a state in the consistent cut. We assume that an internal protocol event, denoted "start CCP," begins each initiator's protocol execution. A proof of correctness of Flood1 is contained in [5]. Flood2 is proved correct almost identically to the protocol in [17].

Protocol 1. Flood1 (FIFO channels, local send inhibition):

- Initiator I, at any time :
  Start CCP; $(*)$ Send(cut,I,j) to each neighbor j before sending new system messages to j.
- Other processors i, immediately upon first receiving a message of the form (cut, k, i) :
  $(*)$ Send(cut, i, j) to each neighbor $j \neq k$ before sending new system messages to j.

Protocol 2. Flood2 (FIFO channels, local receive inhibition):

- Initiator I, at any time :
  Start CCP; Send(cut,I,j) to each neighbor j before receiving new messages. (*)
- Other processors i, immediately upon first receiving a message of the form (cut, k, i) :
  Send(cut, i, j) to all neighbors $j \neq k$ before receiving new messages. (*)

Protocol 3 uses a two-phase method. A spanning tree of the network, assumed to be known in advance, is used to minimize communication. Again there is a distinguished initiator I. Three messages, *PrepareCut, Cut,* and *Resume* are sent respectively down, up, and back down the spanning tree. System send events are disabled as *Cut* messages move up the tree and re-enabled as *Resume* messages move down the tree. Since a message chain from the initiator is required to re-enable the send events, this is an example of global send inhibition. A proof of the correctness of this protocol is contained in [17]. In sum, any inconsistent message sent after the cut—and after the re-enabling of system send events—would have to be received before being sent, due to causal chains from the receiver to the initiator and from the initiator to the sender.

Protocol 3. Tree (non-FIFO channels, global send inhibition):

Let T be a spanning tree of the network rooted at I.
Let $parent(i)$ and $children(i)$ be the parent and children of i in T.
(1) Initiator I: Start CCP; Send $PrepareCut$ to $children(I)$.
(2) Each internal node i, after receiving $PrepareCut$: Send $PrepareCut$ to $children(i)$.
(3) Leaf nodes i, after receiving $PrepareCut$: Disable system sends; Send Cut to $parent(i)$. (*)
(4) Each internal node i, after receiving Cut from $children(i)$: Disable system sends; Send Cut to $parent(i)$. (*)
(5) Initiator I, after receiving Cut from $children(i)$: (*) Send *Resume* to $children(I)$.
(6) Internal nodes i, after receiving *Resume* from $parent(i)$: Send *Resume* to $children(i)$; Enable disabled system sends.
(7) Leaf nodes i, after receiving *Resume* from $parent(i)$: Enable disabled system sends.

Figure 2 illustrates which CCPs are successful under varying system and protocol characteristics. For every case in which none of these protocols can be used, we can demonstrate impossibility as designated by IMP. Specific impossibility results for the IMP† cases are presented in the following section; the remaining impossibilities are immediate consequences of those results.

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

5 Impossibility theorems

In this section we present impossibility results for the IMP† cases in Figure 2. To cover the non-FIFO cases, we assume global receive inhibition and local send inhibition and show that these are not enough to produce cuts in which system messages are guaranteed to be consistent. The remaining non-FIFO cases then follow immediately.

Theorem 1. With global receive inhibition and local send inhibition, there is no CCP for non-

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

Figure 2: Consistent-cut protocols and impossibilities in the inhibition spectrum.

FIFO systems. Moreover, this is true even if we require only that system messages be consistent.

In the proof we will show that, for any CCP as described, there is necessarily a run with the following property: for some two processors, there is causal circularity between the last states preceding their respective cuts in which they are willing to receive messages from each other. We commence by presenting two lemmas from [17]. The first lemma states that the event $receive(m)$ may occur on a processor j as soon as all events of j which happen-before $send(m)$ have occurred.

Lemma 2. Let r be a run such that $l_i \cdot e_i$ is a prefix of r_i and $l_j \cdot send(m)$ is a prefix of r_j, where m is a message from j to i. Suppose that $receive(m)$ is enabled by l_i and that m has not been received in l_i. Then if e_i does not happen-before $send(m)$, there is a run r' such that $l_j \cdot send(m)$ is a prefix of r'_j and $l_i \cdot receive(m)$ is a prefix of r'_i.

If an event e_i of i happens-before an event e_j of j, then if e_j is not a receive, e_i must happen-before the event immediately preceding e_j as well. This is the conclusion of the second lemma.

Lemma 3. Let r be a run, e_i a non-receive event occurring in r_i, and e'_i the event occurring immediately before e_i in r_i. Then for any event e_j, if $e_j \rightarrow e_i$ it must be the case that $e_j \rightarrow e'_i$.

In order to prove Theorem 1, we need only prove that, for any protocol satisfying the conditions of the theorem, in some run of the protocol with respect to some system, the cut produced by the protocol is inconsistent. We can therefore make any worst-case assumptions about the enabling relations on sequences of system events.

Proof of Theorem 1: Given a run r of the protocol, let $l_p \cdot e_p$ be the cut state of p in r for each processor p. Let i be any processor. Let l_i' be the last proper prefix of i's cut state in which receives from some processor are enabled and e'_i be the event following l_i' in r. Let j be any processor whose receives are enabled in l_i'; from the manner in which l_i' was chosen, there must be at least one such processor. Let l_j' be the last proper prefix of j's cut state in which receives from some processor are enabled and e'_j be the event following l_j' in r. Let l_j'' be the last proper prefix of $l'_j \cdot e'_j$ in which receives from i are enabled and e''_j be the event following l_j''. (See Figure 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 $l_i \cdot e_i$. Let $send(m_{ji})$ be a send event from j to i that is enabled by the subsequence of system events in $l_j \cdot e_j$. The protocol may inhibit the occurrence of $send(m_{ij})$ in $l_i \cdot e_i$ and of $send(m_{ji})$ in $l_j \cdot e_j$. However, since this inhibition can only

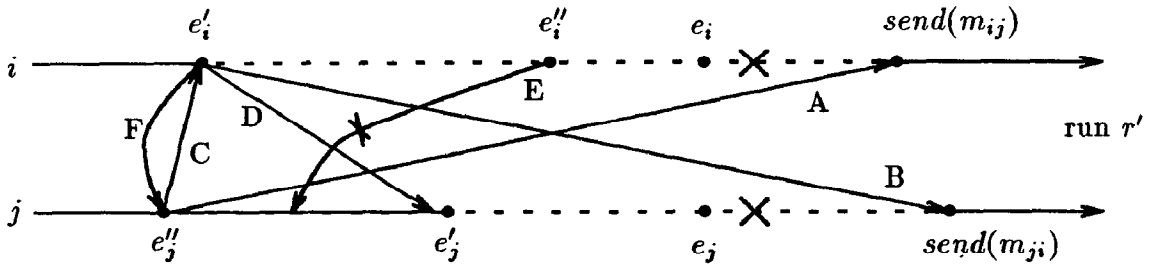

Figure 3: Proof of Theorem 1. Dashed lines are intervals in which no receive events occur, arrows represent happens-before, and 'X' marks cut states. (A),(B) follow from Lemma 2. (C),(D) then follow from Lemma 3. (D) and absence of (E) imply (F).

be local, by Lemma 1 there is a run r' in which $l_p \cdot e_p$ is a prefix of r'_p for each p, and no receive events occur between $l_i \cdot e_i$ (respectively $l_j \cdot e_j$) and the state in which $send(m_{ij})$ (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 r'.

We know that m_{ji} is not received in l'_i; if it were, the cut would be inconsistent. We also know that $receive(m_{ji})$ is enabled by l'_i, by definition of l'_i. Suppose that $e'_i \not\rightarrow send(m_{ji})$. Lemma 2 then implies that there is some run r'' in which $l'_i \cdot receive(m_{ji})$ is a prefix of r''_i. In this run, neither l'_i nor any prefix of l'_i can be i's cut state: if it were, the same local state would be i's cut state in run r' (by condition (2) of the definition of CCP), which is not the case. In sum, r'' is a possible run in which the message m_{ji} is sent after j's cut state and received before i's, and the cut produced by the protocol is inconsistent. Thus the assumption that $e'_i \not\rightarrow send(m_{ji})$ in run r' leads us 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 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 3, 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 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 3). If we can show this, then $e'_i \rightarrow e'_j$ implies $e'_i \rightarrow e''_j$ (arrow (F)) and the partial order will be violated. So 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 $l_i \cdot e_i \cdot \ldots \cdot send(m_{ij})$. In the latter case, we can consider (by Lemma 1) a run which is identical to r' 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 the cut state, 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 3) 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 2 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 is causal circularity in run r'. ∎

Our remaining results deal with FIFO systems. The proofs of these theorems are fairly similar: Theorem 2 is proven in [8]; a slight variation of that argument, presented in the full paper, proves Theorem 3. In both, we assume the existence of a CCP with the required characteristics and derive a contradiction. The proofs differ in that in the non-inhibitory case, one can assume that the event immediately following a cut state is a system send, while in the presence of local send inhibition, one cannot. The proof in [8] produces a possible run in which one of these system messages is inconsistent. The proof of Theorem 3 also produces a possible run in which there is an inconsistent message; however, due to local send inhibition, this message can be made to be a protocol message. This is the intuition behind Flood1.

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

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

6 Conclusions and Related Work

In this work we have distinguished local versus global inhibition for the first time, and consequently defined a spectrum of protocol capabilities with respect to inhibition. We have given a complete analysis of the existence of consistent-cut protocols as a function of these capabilities, while also considering the FIFO or non-FIFO nature of channels and whether or not protocol messages are allowed to be inconsistent with respect to a cut. We have shown that local inhibition is necessary and sufficient to develop a CCP for FIFO systems, while global inhibition of send events is necessary and sufficient for non-FIFO systems.

Many issues are subjects of related research. One such issue is that of piggybacking. Non-FIFO channels in which messages can be "piggybacked" differ from the pure non-FIFO case, because an order is imposed on messages that are packaged together. The "red-white" algorithm of [12,14] is an example of a protocol which piggybacks a protocol marker onto system messages in order to determine consistent cuts. In that protocol, a color—either white or red—is associated with each processor at each point in its local history. A processor starts out white, but may spontaneously turn red, after which it adds a red marker to each message it sends to a neighbor. Any white processor about to receive a red marker turns red immediately before doing so. The white states of the processors form a consistent cut.

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

The red-white protocol, while certainly determining consistent global states, does not satisfy the "distinguishability" requirement (condition (2)) of our definition of a CCP. A particular local state may be the cut state of processor p in one run (because the next event at p in that run is the reception of a red message) but not in another (because the following event is, say, some internal event enabled by that local state). In effect, the protocol designates a *previous* state as a processor's cut state; the processor does not know until after the fact that it was at the cut. Alternatively, one could assume that

a processor is able to "peek" at the contents of arriving messages. Then a processor could, immediately before the reception of a red message, designate its current state as the cut state. This requires stronger capabilities for processors than those granted in our model.

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

Another important issue is the relative complexities of CCPs. It appears that results of [17] can be extended to show that any CCP for FIFO systems using up to local send and receive inhibition requires one message per full-duplex channel. Since CCPs using global send inhibition can use as little as $3(N-1)$ messages, as in Protocol 3, this illustrates a trade-off between global inhibition and message complexity. A lower bound for protocols using global inhibition is unknown. Related work [1] examines message versus time complexity trade-offs in a class of protocols, called *synchronizers*, which resemble CCPs although they differ somewhat in their causal constraints. A message-complexity lower bound for synchronizers is given as a decreasing function of time complexity. However, in that work inhibition is not considered directly, although it is a potential source for increasing the time complexity of protocols.

7 Acknowledgements

We wish to thank Prakash Panangaden for invaluable discussions during the development of this work.

References

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

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

[3] 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.

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

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

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

[7] 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.

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

[9] N. Francez. Distributed termination. *A.C.M. Transactions on Programming Lan-*

guages and Systems, 2(1):42–55, January 1980.

[10] E. Gafni. Perspectives on distributed network protocols: A case for building blocks. In *Proceedings of Milcom '86, Monterrey, CA*, 1986.

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

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

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

[14] 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.

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

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

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

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