The Inhibition Spectrum and the Achievement of Causal Consistency

Carol Critchlow
Kim Taylor*

TR 90-1101
February 1990

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

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

The Inhibition Spectrum and the Achievement of Causal Consistency

Carol Critchlow
Center for Applied Mathematics, Cornell University
cmc@amvax.tn.cornell.edu

Kim Taylor *
Computer Science Department, Cornell University
ket@svax.cs.cornell.edu

February 21, 1990

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 capabilties 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 [10] introduces causality and terms these global states *consistent cuts*. 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 [5], deadlock detection [4], distributed termination [8], and broadcasting [2].

Our spectrum is based on the *inhibitory* capabilities of protocols. Inhibition in asynchronous distributed systems is formally defined in [12]. 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 delays the sending of messages between phases. In [3], this concept is termed *freezing* and is defined for a general notion of superimposed processes. However, no previous work has made the important distinction between *local* and *global* inhibition, as we do, and only [12,7] have proven fundamental relationships between inhibition and determining consistent global states. This paper extends those works significantly.

In [12] 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 [7], it is shown that indeed there is no non-inhibitory CCP for FIFO systems without this atomic receive-send mechanism. 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 from cases where it delays while waiting 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.

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), all previous works 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 prove 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 show 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. For (2) and (3), 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 state in the cut 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 1 of Section 3.)

2 Model and Definitions

We model asynchronous distributed systems, i.e. systems in which (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. 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. Each system consists of a connected network of N processors joined by bi-directional channels. Processors i and j are *neighbors* if they are connected by a channel. 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. 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. 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, and allows us to define formally what it means for a protocol to inhibit the actions of a system. We require of the enabling relation that 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. The alternative assumption is discussed at the end of Section 3.

Given an N-vector $r = (r_1, \ldots, r_N)$, where r_i is a local history of processor i for each i, an ordering on events based on causality is introduced, as in [10]: if e and e' are two events appearing 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 in r_i before e'; (2) e is $send(m)$ and e' is $receive(m)$; or (3) there is an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$. The vector r is a *total run* of the 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) if a send or an internal event of i is forever-enabled [12] in r_i, then it occurs in r_i; and (5) if a receive event of processor i is forever-enabled in r_i, and the corresponding send occurs in the appropriate local history, then the receive occurs 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* in r.

A *protocol* for a system introduces new events, called *protocol events*, and new enabling relations on sequences of system and protocol events. (We separate protocol and system events in order to define inhibition. This model does not allow the merging of system and protocol messages, as in piggybacking techniques. We address this in Section 5.) Runs of this modified system are referred to as *runs of the protocol* with respect to the original system. We require that the projection of a protocol run onto the set of system events be some run of the original system.

An event e_i is *disabled* by a protocol in state l_i of a protocol run if the subsequence of system events in l_i enables e_i in the original system, but l_i does not enable e_i. A protocol is *non-inhibitory* if no event is disabled by the protocol in any protocol run. A non-inhibitory protocol, then, does not interfere with the running of the underlying system. Any protocol which does disable system events is *inhibitory*. We further distinguish two kinds of inhibition.

Definition 1. Suppose event e_p has been disabled by the 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. A protocol is *locally inhibitory* if any event disabled by the protocol in any run of the protocol is delayed locally. An inhibitory protocol that does not have this property is *globally inhibitory.*

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.

Fact 1. Let r be a protocol run, $(l_1, \ldots, l_N)$ a consistent cut in 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 in r', and, for $i = 1, \ldots, k$, there are no receive events at processor p_i between l_{p_i} and the state in which e_{p_i} is re-enabled.

We investigate the existence of *consistent-cut protocols* for our systems.

Definition 2. A *consistent-cut protocol (CCP)* is a protocol which in every system S and every protocol run r, 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; and (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. Furthermore, for every system there is some run of the protocol containing at least one system event.

Thus the protocol creates for each processor i a state in which i "knows" that any messages sent subsequently to j will not be received in j's cut state, regardless of the sequence of steps that j may have taken. We sometimes consider CCPs in which $cut(r)$ may contain inconsistent *protocol* messages.

3 Protocols

In this section we give three CCPs and discuss their inhibitory characteristics. The first two are based on echo algorithms [6]. *Echo1* is essentially the Chandy-Lamport checkpointing algorithm [5]. *Echo2* is similar to that of [12], with the atomic receive-send mechanism replaced by local receive inhibition. The third algorithm, *Tree*, is a two-phase spanning tree protocol from [12]; it is similar to protocols in [8,1]. Properties of these CCPs are summarized below, 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 discussion of message complexity to Section 5.

Name	Channels	Inhibition	Protocol Messages
Echo1	FIFO	local send	inconsistent
Echo2	FIFO	local receive	consistent
Tree	non-FIFO	global send	consistent

In both Echo1 and Echo2, 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 Echo1, 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 Echo2 the cut is reached at the end of the interval. Consequently, it is the reception of system messages which 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. A proof of correctness of Echo1 is contained in [5]. Echo2 is proved almost identically to the echo protocol in [12].

Protocol 1. : **Echo1** (FIFO channels, local send inhibition):

- Initiator I, at any time :

  $(*)$ 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. : **Echo2** (FIFO channels, local receive inhibition):

- Initiator I, at any time :

  Send(cut,I,j) to each neighbor j before receiving any 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 any new messages. $(*)$

Protocol 3 uses a two-phase method. A spanning tree of the network 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 [12].

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: 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 1 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.

		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	Echo2	IMP	Echo2
	Global Receive Inhibition	IMP	Echo2	IMP	Echo2
Local Send Inhibition	No Receive Inhibition	IMP	IMP†	IMP	Echo1
	Local Receive Inhibition	IMP	Echo2	IMP	Echo1,Echo2
	Global Receive Inhibition	IMP	Echo2	IMP†	Echo1,Echo2
Global Send Inhibition	No Receive Inhibition	Tree	Tree	Tree	Tree,Echo1
	Local Receive Inhibition	Tree	Tree,Echo2	Tree	Tree,Echo1,Echo2
	Global Receive Inhibition	Tree	Tree,Echo2	Tree	Tree,Echo1,Echo2

Figure 1: Consistent-cut protocols and impossibilities in the inhibition spectrum.

4 Impossibility Results for Non-FIFO Systems

To cover the non-FIFO impossibilities as illustrated in Figure 1, we assume global receive delay and local send delay (the IMP† case) 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 delay and local send delay, there is no CCP for non-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 a run in which, for some two processors, there is causal circularity between the last states before their respective cuts in which they are willing to receive messages from each other. We commence by presenting three lemmas from [12].

Lemma 1. Any partial run can be extended to a total run.

The second 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 third 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, 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. For any pair of processors p and q, the protocol disables (some subset of) system send events from p to q for some finite number (possibly zero) of local states following $l_p \cdot e_p$. However, since this inhibition can only be local, by Fact 1 there is a run r' in which each processor p still has cut state $l_p \cdot e_p$ (by condition (2) in the definition of CCP), and no receive events occur between $l_p \cdot e_p$ and the state in which any one of these send events is re-enabled by the protocol. We consider r'.

Let i be any processor. Let $l_i{}'$ be the last proper prefix of $l_i \cdot e_i$ in which receives from some neighbor of i are enabled, and let e'_i be the event that follows $l_i{}'$ in run r'. Let j be a processor whose receives are enabled in $l_i{}'$. Let $l_j{}'$ be the last proper prefix of j's cut state in which receives from any processor are enabled and e'_j be the event following $l_j{}'$ in r'. Let $l_j{}''$ be the last proper prefix in which receives from i are enabled and e''_j be the event following $l_j{}''$. (See Figure 2. The receive-free intervals may be empty; e.g., it may be that $e_i = e_i{}'$.)

We can assume that the first event after the re-enabling of i's system send events is a system send $send(m_{ij})$ to j, and that $send(m_{ji})$ occurs similarly at j. By Lemma 2 and condition (2) of the CCP definition, to avoid receiving m_{ji} or m_{ij} before the cut it must be the case that $e'_i \rightarrow send(m_{ji})$ and $e''_j \rightarrow send(m_{ij})$ ((A) and (B) in Figure 2). Since all receives are disabled between e'_j and e_j, and none occur between e_j and $send(m_{ji})$, $e'_i \rightarrow send(m_{ji})$ implies that $e'_i \rightarrow e'_j$ (C). Similarly, $e''_j \rightarrow e'_i$ (D).

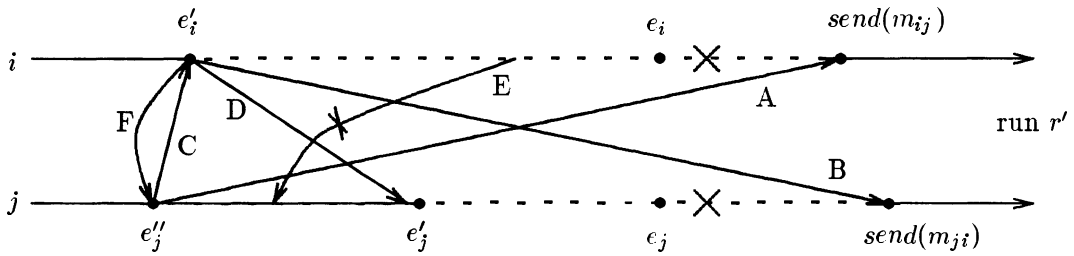

Figure 2: Proof of Theorem 1. Dashed lines are intervals in which no receive events occur, arrows represent happens-before, and 'X' marks cut states. (A),(B) follow from Lemma 2. (C),(D) then follow from Lemma 3. (D) and absence of (E) imply (F).

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 2). If we can show this, then $e'_i \rightarrow e'_j$ implies $e'_i \rightarrow e''_j$ (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. Suppose the first event is $e''_i = send(m)$ to processor k. The event $receive(m)$ must occur in k's cut state, or the remainder of the message sequence will contain an inconsistent message. We can assume that either the event immediately after $send(m_{ij})$ is $send(m_{ik})$ to k, or sends to k have been disabled by the protocol. In the latter case, we can consider (by Fact 1) a run r'' 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 sends to k are re-enabled. Assume that the event following this state is $send(m_{ik})$ to k. In either case, $receive(m) \not\rightarrow send(m_{ik})$ since $receive(m) \not\rightarrow e''_i = send(m)$ and there are no receives between e''_i and $send(m_{ik})$. Using Lemma 2 produces a run in which m_{ik} is inconsistent. Therefore $e'_i \rightarrow e''_j$, and the partial order is violated. ∎

5 Impossibility Results for FIFO Systems

The theorems of this section correspond to the two IMP† results for FIFO systems in Figure 1.

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.

The proofs of these theorems are fairly similar: Theorem 2 is proven in [7]; a slight variation of that argument 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 [7] produces a possible run in which one of these system messages is inconsistent. The proof of Theorem 3, which we present here, 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 Echo1.

We now proceed to the proof of Theorem 3. We assume that there is a CCP for FIFO systems which utilizes local send inhibition only, and through a sequence of lemmas derive a contradiction. Again, we can make worst-case assumptions about the enabling relations on sequences of system events. In particular, for simplicity we assume that all system receives are always enabled.

A sketch of our development is as follows. First we show that, given that event e_i is the last event of i's cut state and that event e_j is in the cut state of j, if $e_i \rightarrow e_j$ then in fact e_i and e_j are the sending and receiving of a single message and e_j is the last event of j's cut state (Lemma 6). Next we show that, when such a message does not occur from i to j, there must be an "essential" message that j sends to i before reaching its cut state; this message always arrives immediately before (in a sense causing) or after the cut state of i (Lemma 7). Next, by Lemmas 8 through 11, we prove the existence of a run in which no cut states occur along a single message as in Lemma 6 and therefore every pair of processors must exchange essential messages; furthermore, for one processor, say k, all of the essential messages which it sends arrive after its neighbors' cut states. Finally, in the final proof we show that there is a possible run as in Lemma 11, but in which k is forced to reach its cut state before sending an essential message to one processor. Absence of this message allows an inconsistent cut to be produced by the protocol.

5.1 Preliminary Lemmas

Lemmas 1 and 3 are used throughout our argument, as is the following lemma from [12], which is the FIFO version of Lemma 2.

Lemma 4. 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, that m has not been received in l_i, and that all messages sent from j to i in l_j are 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.

The next lemma states that if one processor reaches its cut immediately upon (after) sending a message, then if the receiving processor has not yet reached its cut, it must do so upon reception of that message.

Lemma 5. Let r be a run of the protocol, and suppose that j's cut state in r occurs upon the send of a message to i. Then i's cut state in r must occur before the corresponding receive, or immediately after.

Proof: Suppose j's cut state is $l_j \cdot send(m)$, and i's cut state does not occur upon or before $receive(m)$, i.e. there is at least one event e_i after $receive(m)$ and in i's cut state. Let l_i be the prefix of i's local history up to and including $receive(m)$. We assume that system sends to i are enabled by the subsequence of system events in j's cut state. Suppose these send events are disabled by the protocol for some (possibly zero) number of states following j's cut state. By Fact 1, we know there is some run r' with identical cut states and in which there are no receive events between j's cut state and the re-enabling of the system sends. Assume that the event immediately following this re-enabling is a system send to i. Let $send(m')$ be the first send to i occurring after j's cut state (i.e. m' is either a protocol message sent in the receive-free interval or the system message which we have assumed occurs at the end of this interval.) If $e_i \rightarrow send(m')$ then by Lemma 3 and the fact that no receives occur between $send(m)$ and $send(m')$, $e_i \rightarrow send(m)$ and the happens-before partial order is violated. So $e_i \not\rightarrow send(m')$; also, since channels are FIFO, all messages sent from j to i before m' have been received before e_i occurs. Therefore by Lemma 4, there is a run r' in which $l_i \cdot receive(m')$ is a prefix of r'_i, and $l_j \cdot send(m) \cdot \ldots \cdot send(m')$ is a prefix of r'_j. Now i's cut state cannot occur before $receive(m')$ or it would have done so in run r (condition (2) of the definition of CCP). Therefore m' is sent after j's cut state and received before i's, and the cut is inconsistent, which is a contradiction. ∎

The next lemma states the circumstances under which the last event of one cut state can happen-before an event in another cut state.

Lemma 6. Let r be a run of the protocol. Suppose that i's cut state in r is $l_i \cdot e_i$, and $e_i \rightarrow e_j$ where e_j is an event in j's cut state. Then e_i is *send*(m) and e_j is *receive*(m) for some message m, and e_j is the last event of j's cut state.

Proof: First, e_i must be a send to some processor: if not, then the message sequence from e_i to e_j starts after i's cut state but ends before j's, making the cut inconsistent. Therefore e_i is a send.

Suppose e_i is *send*(m), but m is not sent to j. Let k be the processor to which m is sent. Then by Lemma 5, k will either already have reached its cut state when m is received, or will reach its cut upon receiving m. In either case, the remainder of the message sequence to e_j will contain an inconsistent message.

Thus m must be sent to j, and *receive*(m) must occur before e_j, and therefore in j's cut state. Again by Lemma 5, j's cut state must occur upon *receive*(m), so e_j is *receive*(m) and is the last event in j's cut state. ∎

5.2 Technical Lemmas

The question of whether a pair of processors' cut states occur as in Lemma 6, i.e. immediately upon the sending and reception of a single message, is a critical one. If they do, then there is no possibility of an inconsistent message between the processors: any such message would violate either FIFO-ness or the happens-before partial order. If they do not, then there could potentially be an inconsistent message between the processors. We therefore use the following terminology: in a run r, i's and j's cut states *occur along a message* if there is some message m such that i reaches its cut upon sending m and j reaches its cut upon receiving m, or vice versa. A processor j's cut state *occurs along a message* if there is some i such that i's and j's cut states occur along a message.

Lemma 7 states conditions which must be true of the cut states of processors which do not cut along a message.

Lemma 7. Let r be a run of the protocol in which i's and j's cut states do not occur along a j to i message. Let $l_i \cdot e_i$ be i's cut state in r. Then $l_i \cdot e_i$ must contain an event *send*(m) such that, in all runs r' of the protocol in which $l_i \cdot e_i$ is i's cut state and in which i's and j's cut states do not occur along a j to i message, either *receive*(m) is the last event in j's cut state, or *receive*(m) occurs after j's cut state.

Proof: Let j's cut state in run r be $l_j \cdot e_j$ and suppose that there are no sends to j included in i's cut state in r. If sends from i to j have been disabled by the protocol in i's cut state, then by Fact 1 there is a run r' in which cut states remain the same and the re-enabling occurs without intervening receives. If i sends a protocol message to j during the receive-free interval, let m' be the first such message. Otherwise, assume that the first event after the re-enabling is a system send to j, and let m' be the message sent. Either way, there are no receives between e_i and the sending of m'. Now $e_j \not\rightarrow send(m')$: by the preceding observation and Lemma 3, this would imply $e_j \rightarrow e_i$, but by Lemma 6 this could only happen if $e_j = send(m)$ and $e_i = receive(m)$, which we have assumed not to be the case. So $e_j \not\rightarrow send(m')$. Since there are no messages sent from i to j in $l_i \cdot e_i$, the hypotheses of Lemma 4 are satisfied, and therefore there exists a run r'' in which the cut is inconsistent.

So i must send at least one message to j in $l_i \cdot e_i$. Let m be the last such message. Assume there is a run in which i's cut state is $l_i \cdot e_i$, i's and j's cut states do not occur along a j to i message, and $receive(m)$ occurs before e'_j, the last event of j's cut state. As above, if system sends from i to j have been disabled by the protocol in i's cut state, we can consider a possible run r' and message m' such that: r' has the same cut states as r, m' is the first message sent by i after its cut state, and there are no receive events between i's cut state and the sending of m'. Since i's and j's cut states do not occur along a j to i message, $e'_j \not\rightarrow e_i$ so $e'_j \not\rightarrow send(m')$ (since there are no intervening receives). All messages sent to j in $l_i \cdot e_i$ have been received before e'_j, and so applying Lemma 4 produces a run in which the cut is inconsistent. ∎

We call a message such as that described in Lemma 7 *essential* (i.e. an essential message is one upon whose reception the receiving processor will reach its cut, if it has not already). If a message is not essential, but in the run under consideration it is received after or immediately before the receiver's cut, the message is termed *lucky*. We have seen that every pair of neighbors must either have cut states occurring along a message or exchange essential messages in order to guarantee consistency of the cut. Lemma 8 demonstrates how essential messages can be "pushed back" until the receiving processor is forced to reach its cut upon reception of one of them. Later, in order to complete the proof of our theorem, we then demonstrate a run in which this "pushing back" results in a processor reaching its cut state before it has sent all the necessary essential messages. An inconsistent run results.

Lemma 8. Let r be a run of the protocol in which there are processors i and j whose cut states do not

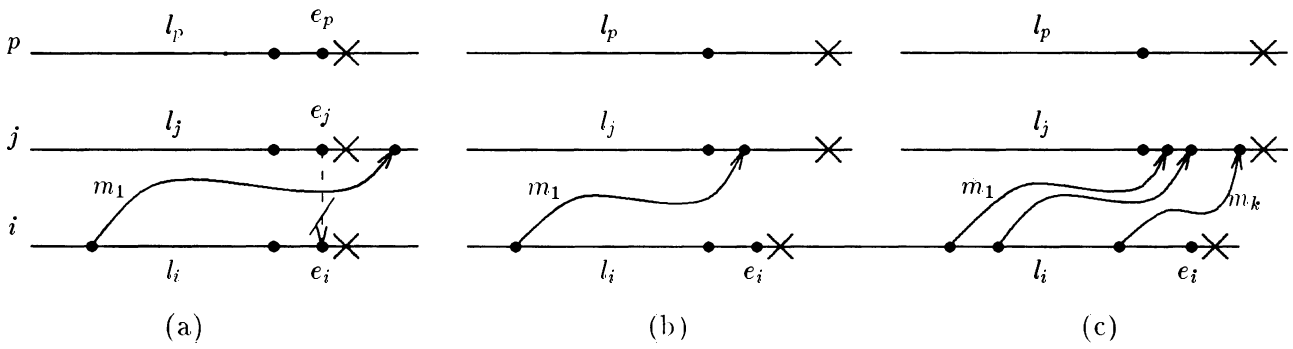

Figure 3: Proof of Lemma 8: (a) the original run r; (b) after the first step, run r^1; (c) the final run r'.

occur along a j to i message. Let $l_p \cdot e_p$ be the cut state of processor p in run r for all p. Then there exists a sequence of messages $m_1, \ldots, m_k$ sent to j in $l_i \cdot e_i$ such that in some run r', $l_j \cdot receive(m_1) \cdot \ldots \cdot receive(m_k)$ is j's cut state. Moreover, it is possible to find an r' for which, in addition, the following two properties hold:

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

2. l_p is a prefix of p's cut state for all p such that $e_j \rightarrow e_p$ (i.e. j's and p's cut states do occur along a j to p message in r).

Proof: Let m_1 be the first message sent to j in $l_i \cdot e_i$ but not received in l_j. Let $m_2, \ldots, m_n$ be any other such messages. By Lemma 7, $m_1, \ldots, m_n$ must contain an essential message.

If $e_j = receive(m_1)$, then r itself satisfies the lemma. Otherwise, let s^1 be the vector of local states where (a) $s^1_j = l_j \cdot receive(m_1)$, (b) $s^1_p = l_p \cdot e_p$ for p as in (1) in the statement of the lemma, and (c) $s^1_p = l_p$ for p as in (2). Then s^1 is a partial run: firstly, each event of each processor is enabled by the preceding local state, since $(l_1 \cdot e_1, \ldots, l_N \cdot e_N)$ is a partial run, and the only additional event is $receive(m_1)$. Secondly, $receive(m_1)$ is the only receive event added, and $send(m_1)$ occurs in $l_i \cdot e_i$; also, the only send event deleted is (possibly) e_j, and any corresponding receives have been removed as well (see (2) in the statement of the lemma). Thus there is no receive event in s^1 for which the corresponding send is missing. Finally, the

happens-before partial order is not violated: by Lemma 6, none of the deleted events can happen-before any of the non-deleted events; and $receive(m_1)$ does not happen-before anything in s^1, so no circularity is introduced. FIFO-ness is not violated either, as m_1 is the first i to j message not received in l_j.

Thus s^1 is a partial run, and can be extended by Lemma 1 to a total run r^1. (See Figure 3 (b). Event e_p in (a) follows l_p in (b) if $e_j \not\rightarrow e_p$.) If j's cut state in r^1 is $l_j \cdot receive(m_1)$, then r^1 satisfies the lemma. Otherwise, let s^2 be the vector of local states such that (a) $s_j^2 = l_j \cdot receive(m_1) \cdot receive(m_2)$, (b) $s_p^2 = l_p \cdot e_p$ for p as in (1), and (c) $s_p^2 = l_p$ for p as in (2). The same argument as above proceeds. Since $m_1, \ldots, m_n$ contain an essential message m_k, we can repeat this process to obtain a run satisfying the lemma, in k or fewer steps (Figure 3 (c)). ∎

The following three lemmas show that there exists a run in which one processor sends an essential message to each of its neighbors, but all neighbors reach their cut state before that message is received. The final proof will then show how this run can be manipulated to form a possible run in which the cut is inconsistent. We start by showing that it is possible to find a run in which at least one processor's cut state ends with a receive event.

Lemma 9. There exists a protocol run r in which there is a processor j such that $degree(j) \geq 2$ and j's cut state in r occurs upon a receive event.

Proof: Let S be any system such that the processor network contains two neighboring processors i and j each with degree at least two. Suppose there exists a run r of the protocol with respect to S in which neither i's nor j's cut state occurs upon a receive; therefore, they do not occur along a j to i message. Lemma 8 can then be applied, and produces a run in which j's cut state occurs upon receipt of a message from i. ∎

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

Lemma 10. There exists a protocol run in which there is a processor whose cut state occurs upon reception of a message but not along a message (i.e. the last event of one cut state is $receive(m)$ but $send(m)$ is not the last event of any cut state).

Proof: By Lemma 9 there is a run r and a processor j of degree at least two such that j's last cut-state event in r is a receive. Let $l_j \cdot e_j$ be j's cut state. Let i be a neighbor of j such that e_j is not the receive of a message from i; it is possible to find such an i since $degree(j) \geq 2$. By Lemma 7 there is an essential message sent to i in l_j (since e_j is not a send). Let m be the first j to i message that is lucky or essential. If i's last cut-state event e_i is $receive(m)$ then i satisfies the lemma. If not, one can apply Lemma 8 to get a run in which i's cut state does occur upon reception of a message from j and in which $l_j \cdot e_j$ is j's cut state. In this new run, i satisfies the lemma. ∎

The next step is to find a run in which each processor's last cut-state event is the receive of a message, and there is one processor which has sent none of these messages.

Lemma 11. There exists a protocol run in which (a) every processor reaches its cut upon a receive but not along a message; and (b) there is a processor j such that none of j's neighbors reaches its cut upon receiving a message from j.

Proof: By Lemma 10, there is a protocol run r in which at least one processor i satisfies the condition mentioned in (a). If i's last cut-state event is $receive(m)$, let p be the processor in whose local history $send(m)$ occurs. Set $D = \{i\}$, and let Q be a queue consisting of all neighbors of i, with p at the head. Since i's and p's cut states do not occur along a p to i message, there exists (by Lemma 8) a run r^1 in which p's cut state occurs upon reception of a message from i, and in which i's cut state remains unchanged. Add p to D, delete it from Q, and add its neighbors other than i to the back of Q. Repeat this procedure: at any step, the processor p' at the head of Q is a neighbor of a processor $k \in D$, and so p''s and k's cut states do not occur along a message. Lemma 8 gives a run r^n of S in which p''s cut state occurs upon reception of a message from k, and for all $d \in D$, d's cut state in r^n is identical to that in r^{n-1}. Processor p' satisfies the condition of (a) in run r^n; add it to D, delete it from Q, and add all its neighbors which are neither in D nor already in Q to Q. Since at each step one processor is added to D, this procedure must eventually halt. Since the network is connected, all processors will eventually be added to Q and thus to D. Therefore the repetition ends with a run r^n satisfying (a).

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

5.3 Proof of Theorem 3

By the final lemma above, we can find a run r and a processor j such that j's cut state in r occurs upon a receive but not along a message, and such that all of j's neighbors reach their cuts upon reception of messages sent by processors other than j. Let j's cut state in run r be $l_j \cdot e_j$. Since j's cut state does not occur along a message, and e_j is a receive event, l_j must contain the send of an essential message to every neighbor of j. Let i be the processor whose first lucky or essential message from j is sent last in l_j. Let $l_i \cdot e_i$ be i's cut state in r. (We know that e_i is a receive, but not of a message from j.) Let $send(m)$ be the first send of a lucky/essential message to i in l_j. Then neither $send(m)$ nor any event after it can happen-before any event in any other processor's cut state: if l_j' is the portion of j's cut state before $send(m)$, then $l_j' \cdot send(m)$ contains the send of a lucky/essential message to each of j's neighbors. By assumption, no processor reaches its cut upon receiving a message from j, so each of these messages is received after the receiver's cut state. Thus m will also only be received after the receiver's cut state, and any message sequence starting at or after $send(m)$ to an event of another processor's cut state will make the cut inconsistent. Therefore $send(m)$ does not happen-before any event in another processor's cut state, and no event which follows it in j's local history can do so either.

Let $m_1, \ldots, m_n$ be the messages sent from i to j in l_i that are not received in l_j' (there is at least one such by Lemma 7). The above argument shows that $send(m) \not\rightarrow send(m_1)$. By an argument almost identical to that in Lemma 8, we can find a run r' in which every processor $p \neq j$ has the same cut state as in r, and $l_j' \cdot receive(m_1) \cdot \ldots \cdot receive(m_k)$ is j's cut state, for some $k \leq n$. (The only difference in the argument is the justification for the assertion that "none of the deleted events can happen-before any of the non-deleted events". In the proof of Lemma 8, we invoke Lemma 6; here we use the argument in the last paragraph above.)

As in the proof of Lemma 7, if system sends from j to i have been disabled by the protocol in j's cut state in r', we can consider a possible run r'' and message m' such that: r'' has the same cut states as r', m' is the first message sent by j after its cut state, and there are no receive events between j's cut state and the sending of m'. Since e_i is a receive event, it does not happen-before any event in j's cut state (Lemma 6). There are no receive events between j's cut state and $send(m')$, so e_i does not happen-before $send(m')$ either. Therefore the hypotheses of Lemma 4 are satisfied, and there is a possible run in which

message m' is inconsistent with respect to the cut formed by the protocol. ∎

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 on 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 sends is necessary and sufficient for non-FIFO systems.

Many issues are subjects of related research. 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. There are CCPs which use piggybacking [9,11]: however, preliminary investigation shows that they in turn require other strong capabilities: either the ability to "peek" at message contents before reading, or to designate a past local state as the cut state. Another approach is to simulate FIFO channels by including the entire message history with every message; this in turn requires unbounded message size.

Another important issue is the relative complexities of CCPs. It appears that results of [12] can be extended to show that any CCP for FIFO systems using up to local send and global 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 send inhibition and message complexity. A lower bound for protocols using global send inhibition is unknown. Related work [1] examines message versus time complexity trade-offs in a class of protocols, called *synchronizers*, which resemble CCPs. A message-complexity lower bound for synchronizers is given as a decreasing function of time complexity.

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] 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.

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

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

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

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

[11] 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.

[12] 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.