The Role of Inhibition in Asynchronous Consistent-Cut Protocols

Kim Taylor*

TR 89-995
April 1989

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

*This research was supported by an AT&T Ph. D. Scholarship.

The Role of Inhibition in Asynchronous Consistent-Cut Protocols

Kim Taylor *

Computer Science Department, Cornell University
4130 Upson Hall, Ithaca, NY 14850
ket@svax.cs.cornell.edu

April 28, 1989

Abstract

We present results relevant to the development of *consistent-cut* protocols. Consistent-cut protocols are those which are based on finding a *consistent global state* in an underlying distributed computation; they are used for a variety of applications such as system checkpointing and deadlock detection. We formally define what it means for a protocol to be *non-inhibitory*, which intuitively means that it does not prevent any actions from occurring in an underlying system computation. We prove that there is no non-inhibitory consistent-cut protocol for non-FIFO asynchronous systems. We also give a lower bound on communication for non-inhibitory consistent-cut protocols for FIFO systems of one message per bidirectional channel (up to $\frac{1}{2}(n^2 - n)$, for completely connected networks). We present two protocols, one non-inhibitory requiring up to two messages between each pair of neighboring nodes in a network and the other inhibitory and requiring only $3(n - 1)$ messages total. In most networks these results illustrate a tradeoff between the amount of necessary communication and the willingness to inhibit actions of the underlying system. Additionally, our inhibitory protocol also works for non-FIFO systems, thus illustrating that the inhibitory condition is exactly what is required to develop consistent-cut protocols for non-FIFO systems which satisfy our model.

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

1 Introduction

In this paper we investigate the problem of developing *consistent-cut* protocols in asynchronous systems. Consistent-cut protocols are based on finding a *consistent global state* in a underlying computation, i.e., a set of local states that do not directly depend on one another and may occur simultaneously. Consistent-cut protocols are very common in asynchronous systems [4] and have applications in checkpointing and rollback recovery [9], deadlock detection [3], and distributed databases [2].

We consider whether or not consistent-cut protocols are *non-inhibitory*, i.e. do not prevent operations which would otherwise take place in the underlying system. For example, non-inhibitory protocols cannot delay the sending and reception of messages which would take place if the protocol were not being run. We have observed that a variation on the "echo" algorithm of Chang [6] and on the checkpointing algorithm of Chandy and Lamport [4] - which does not delay actions of the underlying system - finds a consistent cut in systems with first-in-first-out channels but sends up to two messages between each pair of connected nodes in a system. On the other hand, a two-phase commit variant suspends underlying system send events for an interval but requires only $3(n-1)$ messages total, which is substantially less communication in most networks. Whether or not the less expensive algorithm is indeed better depends upon the degree of degradation caused by a suspension of send events in the underlying application.

This paper presents the following results. (1) There is no non-inhibitory consistent-cut protocol for non-FIFO asynchronous systems. (2) Any non-inhibitory consistent-cut protocol for FIFO asynchronous systems requires one message on each pair of channels between neighboring nodes. (3) There is a non-inhibitory consistent-cut protocol for FIFO asynchronous systems using up to two messages between neighboring nodes. (4) There is an *inhibitory* consistent-cut protocol for both FIFO and non-FIFO asynchronous systems using exactly $3(n-1)$ messages, where n is the number of nodes in the network.

For most FIFO networks (all but very sparsely connected ones) results (2) through (4) illustrate a tradeoff between the amount of communication necessary in consistent-cut protocols and whether or not they inhibit actions of the underlying system. In particular, this is true for completely connected networks. Also, results (1) and (4) indicate that the property of inhibition is exactly what is required for a non-FIFO network to achieve a consistent cut in the systems that satisfy our model.

The paper is organized as follows. Section 2 contains our system model and formal definitions for non-inhibitory and consistent-cut protocols. Section 3 contains lemmas necessary for subsequent theorems. In section 4 we prove our impossibility and lower-bound results. Section 5 contains the non-inhibitory and inhibitory consistent-cut protocols. Finally, in section 6 we discuss our conclusions, some related work, and potential future work.

2 Model and Definitions

2.1 System Model

Our goal is to model reliable asynchronous distributed systems. By asynchronous, we mean: (1) there is no global clock, (2) there are no bounds on message transmission time, (3) there are no assumptions made concerning the relative speed of processors, and (4) the sending and arrival of a message are independent, i.e., there is no CSP-style synchrony of the two events. By reliable we mean that processes do not fail and that the communication medium eventually delivers all messages sent, message contents are not permuted, and no messages are delivered that were not sent. We also assume that communication channels are bidirectional and that the network is connected, so that there is at least one communication path from each node to every other node.

The model is very similar to that of Chandy and Misra [5] and to that used in our related work [11]. A system is characterized by a set of possible executions, or *runs*. Runs are composed of sequences of *events* corresponding to the local operations of individual processes. Unlike these models, we explicitly include the *enabling* of events by past event sequences [12] and we explicitly separate protocol events from underlying system events. These two features allow us to define *non-inhibitory* protocols as those whose events never disable those of the underlying system. We also include mechanisms for sending multiple messages atomically (in one indivisible process operation) and for receiving a message and sending responses in one atomic operation. (Most of our results - the only exception being theorem 3, the existence of a non-inhibitory consistent-cut protocol for FIFO systems -

do not require these primitives. Our impossibility and lower bound results immediately generalize to a weaker model without them, and they are not necessary to prove correctness of our inhibitory algorithm (theorem 4). This is discussed further in the section on protocols and in our conclusions.)

The basic entity in our system model is the *event.* Let $\mathcal{I} = 1, 2, ..., N$ be process identifiers. Then $\mathcal{E} = (E_1, E_2, ..., E_N)$ is a vector of sets of possible events, where events in set E_i correspond to operations performed by process i. There are four types of events: (1) *sends*, (2) *receives*, (3) *receive-sends*, and (4) *internal events*. A send event is parameterized by a set of up to N *messages* (maximum of one per process). Each message is simply a string of bits, a unique sequence number, and two process identifiers indicating the sender and receiver of the message. A receive event is parameterized by a single message. A receive-send is parameterized by one to $N+1$ messages, the first corresponding to a received message and the remainder corresponding to a (possibly empty) set of messages sent in the same atomic step.

A *local history of* i is any sequence of events, denoted $e_1^i, e_2^i, ...$, such that each element of the sequence is a member of the set E_i. A *local state* is any finite prefix of a local history. $Local\_states_i$ is the set of all possible local states of E_i. The null *initial state* is an element of $Local\_states_i$ for all i. Concatenation of events to produce new local states is denoted by "$\cdot$" as in $l = l' \cdot e$. If local state s is a prefix of local history h, we say that predicate $Prefix(s, h)$ is true.

Our model also includes $\mathcal{M} = (M_1, ..., M_N)$, a vector of *enabling relations.* An enabling relation M_i specifies that events may take place based on prerequisite events. Specifically, M_i is an *enabling relation on* E_i iff it satisfies

$$M_i \subseteq Local\_states_i \times E_i.$$

If $(l_i, e_i) \in M_i$ then we say l_i *enables* e_i. An enabling relation is *finite-branching* iff it contains a finite number of events enabled by each local state. We will require that our enabling relations be finite-branching.

Finally, a *channel* set $\mathcal{C}$ is a set of unordered pairs of elements from $\mathcal{I}$. The channel set represents the bidirectional communication links of a system. Pair (i, j) is in the set iff process i can send a message directly to process j and vice-versa.

We can now define our system model. A *system* $\mathcal{S}$ is a quadruple

$(\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$ consisting of a process identifier set $\mathcal{I} = 1, ..., N$, a channel set $\mathcal{C}$, an N-vector of event sets $\mathcal{E}$, and an N-vector of enabling relations $\mathcal{M}$ such that (1) for each process i, M_i is a finite-branching enabling relation on 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 $(i, i_1), (i_1, i_2,), ..., (i_k, j)$ in $\mathcal{C}$ (the network is connected).

Given a system $\mathcal{S}$, we next want to model the *runs*, or possible executions, of the system. In order to guarantee that these executions correspond to those of a potentially-real distributed system, we impose a temporal ordering on the events in them. This temporal ordering is based on *causality* as introduced by Lamport [10]. We define two relations on vector of N of local histories, one per process. First, e_1^i *happens-immediately-before* e_2^j, denoted $e_1^i \mapsto e_2^j$, iff (1) i equals j and e_2^j is the next event after e_1^i in the local history of i (j), or (2) e_1^i includes the sending of a message and e_2^j includes its reception. Next, *happens-before*, denoted $\rightarrow$, is the irreflexive transitive closure of happens-immediately-before. Therefore, if $e_1^i \rightarrow e_2^j$ then either (1) $e_1^i \mapsto e_2^j$, or (2) there exists an event e_3^k such that $e_1^i \rightarrow e_3^k$ and $e_3^k \rightarrow e_2^j$.

We can now define the runs of a system. First of all, a *partial run* of a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$ is a vector of N local states, one per process, such that (1) for every receive there is a corresponding send, (2) $\rightarrow$ is a partial order on the set of events, (3) for all events there is an enabling state (possibly the null initial state) immediately before it in the local state. An *initial partial run* is composed solely of null initial states.

We say that an event e is *forever-enabled* in a local history if it is enabled by some local state l in the history (i.e. $(l, e) \in \mathcal{M}$), it is not in the history (i.e. it is never executed), and it is enabled by all local states for which l is a prefix. A *total run* of system $\mathcal{S}$ is a vector of local histories that satisfies conditions (1)-(3) on partial runs above and, in addition, satisfies: (4) there are no forever-enabled send or internal events, and (5) there are no forever-enabled receive or receive-send events for which there is a corresponding send event. We will often refer to a total run as simply a "run." We let $r[i]$ denote the local history of i in run r. We say that run r *extends* partial run r' iff $\forall i\, .\, \mathit{Prefix}(r'[i], r[i])$.

We often consider the special case of *first-in-first-out (FIFO)* systems, in which messages are guaranteed to arrive in the order in which they are sent. For such systems we add an additional constraint on total runs: for any two messages m_1 and m_2, both from process i to process j, if

$send(m_1) \rightarrow send(m_2)$ then $receive(m_1) \rightarrow receive(m_2)$.

2.2 Protocols

In this section we will formally define protocols, non-inhibitory protocols, and consistent-cut protocols. We define protocols in an additive fashion, so as to separate the activities of the protocol from those of the underlying system. Obviously in our definition of non-inhibitory we will want to restrict the extent to which the protocol affects system events. This is true, to a lesser extent, of our general protocol definition as well, since we desire that any protocol allow the system to accomplish its original tasks as well as the protocol task. For example, we would not want to consider a consistent-cut protocol in which each process simply halts after sending its first message. We will require that the projection of the system events in the run of a protocol be some valid run of the original system.

Our protocols will map a system $\mathcal{S} = (\mathcal{I}, \mathcal{C}, \mathcal{E}, \mathcal{M})$ to a new system $\mathcal{P}(\mathcal{S}) = (\mathcal{I}, \mathcal{C}, \mathcal{E}', \mathcal{M}')$ under certain constraints. The events in $\mathcal{E}$ are called the *system events* and the events in $\mathcal{E}' - \mathcal{E}$ are called the *protocol events.* Given a sequence composed of both system and protocol events, the function *SysEvents* maps it to a new sequence containing only the system events in their original order.

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' is an enabling relation on E_i', and (3) for all total runs $r' = (h_1', ..., h_N')$ of $\mathcal{P}(\mathcal{S})$, where h_i' is the local history of process i, run $r = (SysEvents(h_1'), ..., SysEvents(h_N'))$ is a total run of the original system $\mathcal{S}$. Note that the possible projected runs of the underlying system may only be a subset of the original possible runs; for example at points where the original protocol may have had a choice between performing internal events and send events it may be forced to perform only internal events. Also, the underlying system can be forced to forego system activity for any number of protocol steps, as long as eventually a valid system run is completed. Such "inhibitions" will not be possible for non-inhibitory protocols.

A protocol is *non-inhibitory* iff for all local states l and events e

$$\text{if } (SysEvents(l), e) \in \mathcal{M} \text{ then } (l, e) \in \mathcal{M}'.$$

In other words, if a sequence of system events enables some system event

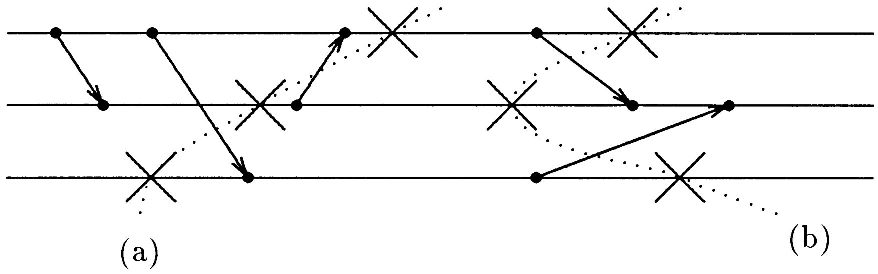

Figure 1: Inconsisent (a) vs. consistent (b) cuts.

in the original system, then the same sequence with protocol events interspersed will enable that system event in the protocol.

Given a run r, a prefix of the run (a vector of prefixes of local histories) is typically called a *consistent cut* if it is also a partial run. In other words, a *consistent cut* of run r is a vector of local states of r, denoted $cut(r) = (l_1, ..., l_N)$, such that for any message m from i to j if $receive(m)$ is in l_j then $send(m)$ is in l_i. (See figure 1. The states of (b) form a consistent cut whereas those of (a) do not.)

For simplicity we assume that consistent-cut protocols create only one cut in the system; our results can be easily extended to the multiple-cut case. Formally, a *consistent-cut protocol* is one in which every run r in $\mathcal{P}(\mathcal{S})$ contains a unique set of local states $cut(r) = (l_1, ..., l_N)$ such that (1) $cut(r)$ is a consistent cut, not equal to the initial partial run, and (2) for all processes i, in any run r' containing l_i, l_i is in $cut(r')$. The second condition implies that the occurrence of the cut is deterministic with respect to the sequence of local events leading up to it, i.e. if process i's cut state immediately follows sequence of events l in one run and the exact same sequence l occurs in a different run, then i's cut state in the later run will immediately follow the same sequence. This is necessary if some local action is to be immediately performed as a result of the occurrence of the cut; for example, taking a checkpoint. (One can devise algorithms which determine the existence of a cut "after the fact" which do not satisfy condition (2)[7]. However this may be inappropriate or cause undesirable overhead in particular applications. In the checkpointing example, it necessitates taking multiple checkpoints, in order to encompass all of the local states which may later be included in the system checkpoint. This is undesirable due to the expense of taking a checkpoint.) As noted in the introduction, consistent-cut protocols encompass a wide range of applications and are therefore an important class to study.

3 Lemmas

In this section we present four lemmas which will be useful in the theorems that follow. The first says that any partial run can be extended to a total run.

Lemma 1 *If r is a partial run of $\mathcal{S}$, then there exists at least one total run r' which extends it, i.e. $\forall i\,.\, Prefix(r[i], r'[i])$.*

Proof: We construct a (possibly infinite) total run r' from r by putting newly enabled events onto several FIFO queues and iteratively deleting events from the queues as they are added to the run. Initialize r' to r. For each process i, let Q_i^0 be a queue of send and internal events which are enabled by local state $r'[i]$. Similarly, for each process j which is a neighbor of i, let Q_i^j be a queue of enabled receive (or receive-send) events for which there is a corresponding send event in r'. The queues are initialized by adding the appropriate events in a random order.

We iterate through processes one at a time, and within each process we visit its queues one at a time. For each process-queue pair, if the queue is non-null then we add the first event on the queue to run r'. That event is deleted from the queue, along with any events on any of the process' queues which are no longer enabled by the resulting local state. Any events which are enabled by the new local state but were not previously enabled are added to the appropriate queue. Newly enabled events are added to queues in any order (but they come after events previously on the queue).

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

It is now only necessary to show that conditions (4) and (5) on total runs are satisfied, i.e. that there are no forever-enabled send or interal events and that there are no forever-enabled receive (or receive-send) events for which there is a corresponding send event. If there were such events they would remain on a queue indefinitely. However, all queues are finite since we require that our systems contain only finite-branching enabling relations. Also, there are a finite number of queues to be visited on each iteration. Since events are removed from finite queues in a FIFO manner, any event remaining on a queue must eventually be added to r' and deleted from the queue. Hence, no event is forever-enabled. This completes the proof. ∎

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

Lemma 2 *Let run r of (non-FIFO) system $\mathcal{S}$ contain events e_j on process j and $send(m)$ on process i, where m is from i to j, i.e. there are local states l_i and l_j such that*

$$Prefix(l_i \cdot send(m), r[i]) \text{ and } Prefix(l_j \cdot e_j, r[j]).$$

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

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

$$Prefix(l_i \cdot send(m), r'[i]) \text{ and } Prefix(l_j \cdot receive(m), r'[j]).$$

(See figure 2. The crossed solid line indicates "not happens-before" and the dashed lines connect the sending and reception of m.)

Proof: The proof procedes by first showing that there is a partial run r'' of $\mathcal{S}$ where $Prefix(l_i \cdot send(m), r''[i])$ and $r''[i] = l_j$ (without e_j). Then $receive(m)$ is added to r'' still resulting in a partial run which can be extended to a total run r' by lemma 1.

We find a partial run of r, which satisfies the conditions on r'' above, iteratively. (It is not obvious that such a partial run exists. Perhaps any combination of local states having the required prefixes for i and j will have a message m - potentially involving other processes - which is received

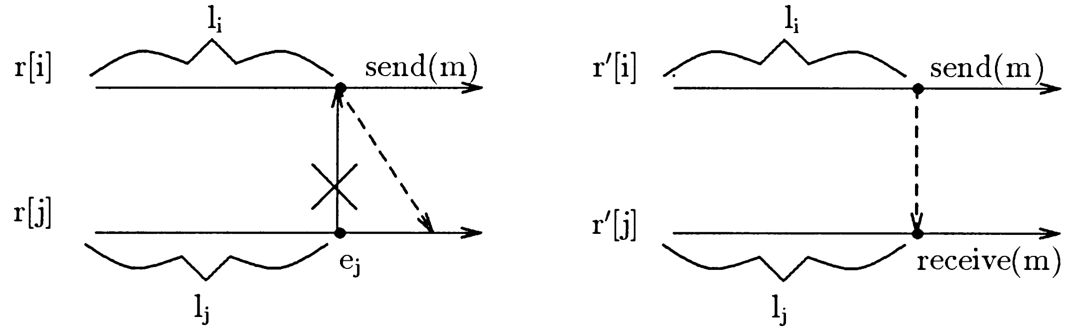

Figure 2: Lemma 2.

but not sent.) Let $MinSends(l_x, y)$ be the minimum local state of process y in run r which includes the sending of all messages received by x from y in local history l_x. Let $max(l_1, l_2)$, where l_1 and l_2 are local states of a single process in a single run, be the largest sequence of the two (and therefore encompassing the other). Initialize a vector of local states s as follows: $s[j] = l_j$, $s[i] = max(l_i \cdot send(m), MinSends(l_j, i))$, and $s[x] = MinSends(l_j, x)$ for x not equal to i or j. Of course this initial set of states is not necessarily a partial run. On each step of the iteration, find a message m' from any process a to any other process b whose reception is in s but whose sending is not (if such a message doesn't exist then we are finished). Add events from $r[a]$ to $s[a]$ until the necessary send is included.

We next make an observation to be used extensively in the remainder of the proof. Consider any message m' causing inconsistency above. Either $receive(m')$ is in the original cut, or is included as the result of adding states until the sending of some m'' is included, so $receive(m') \rightarrow send(m'')$. This argument can be continued resulting in a chain of messages

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

where $receive(m^0)$ is in the original set of local states s. (See figure 3. The solid circles represent the original set of states.)

We now show that (1) the iteration terminates, and (2) the local state of j is not changed, i.e. upon termination $s[j] = l_j$. Suppose that the iteration above never terminates. Then there must be message chains as

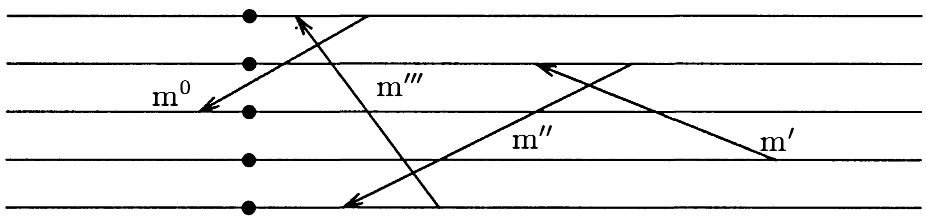

Figure 3: Proof of Lemma 2: Message chains during iterative construction.

above where two messages m_1 and m_2 are sent by the same process. If $m_1 \rightarrow m_2$ in the chain then also $m_2 \rightarrow m_1$ because there is a local state which includes m_2 but not m_1. This obviously cannot occur in any run. Therefore the iteration terminates.

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

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

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

Let r'' be the final s of the iteration. Then r'' is a partial run of $\mathcal{S}$ such that $r''[j] = l_j$ and $Prefix(l_i \cdot send(m), r''[i])$. Since $receive(m)$ is enabled by l_j in system s, adding event $receive(m)$ to $r''[j]$ still results in a partial run of $\mathcal{S}$. By lemma 1 this can be extended to a full run r' as the theorem states. ∎

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

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

$$\mathit{Prefix}(l_i \cdot send(m), r[i]) \text{ and } \mathit{Prefix}(l_j \cdot e_j, r[j]).$$

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

If (1) $e_j \not\to$ (does not happen before) $send(m)$ and (2) there is not a message m' from i to j sent before m but not received in l_j, then there exists a run r' also in $\mathcal{S}$ such that

$$\mathit{Prefix}(l_i \cdot send(m), r'[i]) \text{ and } \mathit{Prefix}(l_j \cdot receive(m), r'[j]).$$

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

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

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

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

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

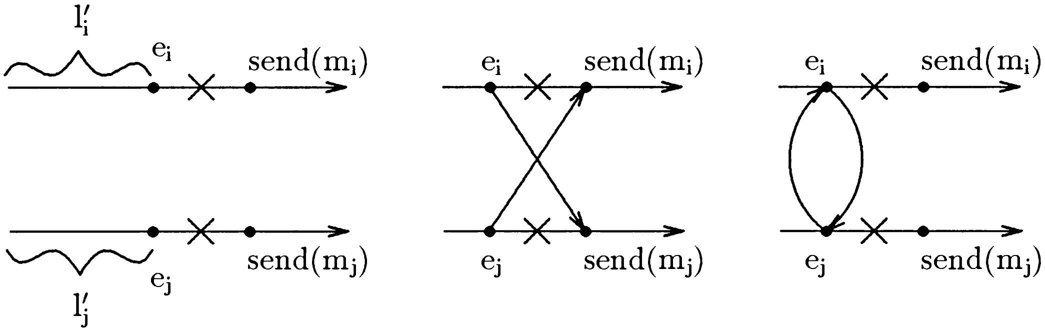

Figure 4: Proof of Theorem 1. (a) Run r. (b) From lemma 2. (c) From lemma 4.

4 Impossibility and Lower-Bound Theorems

In this section we prove two results regarding non-inhibitory consistent-cut protocols: that one does not exist for non-FIFO systems, and that any for FIFO systems requires one message between each pair of directly connected nodes.

Theorem 1 *There is no non-inhibitory consistent-cut protocol for non-FIFO systems.*

Proof: Suppose that there is such a protocol $\mathcal{P}$, and let $\mathcal{S}$ be any non-FIFO system. For any run r in $\mathcal{P}(\mathcal{S})$, there must be a set of local states $cut(r) = (l_1, ... l_N)$ as in the definition of a consistent-cut protocol. Let run r be a run in $\mathcal{P}(\mathcal{S})$ in which there is a system send event of a message m_i from i to j immediately after l_i and similarly a send of m_j from j to i immediately after l_j. Such system events cannot be disabled, by the definition of a non-inhibitory protocol. Let $l_i = l'_i \cdot e_i$ and $l_j = l'_j \cdot e_j$. (See figure 4(a). The '$\times$' indicates the local states of the cut.) Also let the receptions of messages m_i and m_j be enabled by local states l'_j and l'_i, respectively. Again, a non-inhibitory protocol cannot prevent such enablings.

Suppose $e_j \not\rightarrow send(m_i)$. Then by Lemma 2 there exists a run r' in $\mathcal{P}(\mathcal{S})$ such that

$$Prefix(l_i \cdot send(m_i), r'[i]) \text{ and } Prefix(l'_j \cdot receive(m_i), r'[j]).$$

By the definition of a consistent-cut protocol, there must be a cut $cut(r') = (k_1, ...k_N)$. However, k_j cannot be in l'_j or a prefix of it; otherwise the local state of j in $cut(r)$ would not be unique by condition (2) of the definition. Therefore the reception of m_i is in $cut(r')$ but the sending is not. This contradicts the assumption that $\mathcal{P}$ is a consistent-cut protocol. Hence the assumption that $e_j \not\rightarrow send(m_i)$ must be false. A symmetric argument holds for e_i and $send(m_j)$. So we have

$$e_j \rightarrow send(m_i) \text{ and } e_i \rightarrow send(m_j).$$

(Figure 4(b).) Since $e_j \mapsto send(m_j)$ and $e_i \mapsto send(m_i)$, it follows from Lemma 4 that $e_i \rightarrow e_j$ and $e_j \rightarrow e_i$. (Figure 4(c).) Therefore r cannot be a run of the system since the happens-before partial order is violated. ∎

Theorem 2 *Any non-inhibitory consistent-cut protocol for FIFO systems requires one message between each pair of neighboring nodes.*

Proof: We assume that one message between each pair of neighboring nodes is not necessary and arrive at a contradiction. Suppose that P is such a protocol and sends no mesages between nodes i and j.

The contradiction is similar to that of theorem 1; we refer to it for brevity. Let run r of $P(S)$ with $cut(r) = (l_1, ...l_N)$ send messages m_i, m_j immediately after l_i, l_j as before, with the receptions of m_i and m_j enabled in l'_j and l'_i, respectively. Additionally assume that run r contains no system messages sent by i to j before l_i or by j to i before l_j.

In run r there is no message m'_i satisfying $send(m'_i) \rightarrow send(m_i)$ but $receive(m'_i)$ is not in l_j, as in condition (2) of lemma 3. It then follows from arguments identical to those of theorem 1 (using lemma 3 rather than lemma 2) that $e_j \rightarrow send(m_i)$ and symmetrically $e_i \rightarrow send(m_j)$. The remainder of the proof follows theorem 1 exactly. ∎

5 Protocols

In this section we present two consistent-cut protocols; one non-inhibitory and the other inhibitory.

Theorem 3 *There is a non-inhibitory consistent-cut protocol for FIFO systems using up to two messages between each pair of neighboring nodes.*

The algorithm is a variant of the echo algorithm due to Chang [6] and of the Chandy-Lamport checkpointing algorithm [4]. Messages are sent along every channel in the system, beginning with a distinguished initiator I. A consistent-cut occurs because any message sent after the algorithm messages must arrive after them due to FIFO channels. The local states forming the cut are those immediately following the sending of the algorithm messages, as indicated by the "Cut" comments.

Algorithm 1 *Non-inhibitory Consistent-Cut Protocol*

- Initiator I: Send(cut,I,j) to all neighbors j. {Cut}
- Other processes i, immediately (atomic receive-send) upon first receiving a message of the form (cut, k, i): Send(cut, i, j) to all neighbors $j \neq k$. {Cut}

Proof of Theorem 3: Clearly algorithm 1 is non-inhibitory. We prove that the indicated states form a consistent cut in FIFO systems similarly to [11] and [4]. Suppose they do not. Then there is some message m, say from i to j, which is sent after i's *Cut* state but received before j's. Before reaching its *Cut* state, by the algorithm either (1) i had previously received a *cut* message (call it m_j) from j, or (2) i sent a *cut* message (call it m_i) to j. In case (1), j must have reached its *Cut* state immediately after sending the message. The reception of m before this would cause circularity in $\rightarrow$, because it implies

$$send(m_j) \rightarrow receive(m_j) \rightarrow send(m) \rightarrow receive(m) \rightarrow send(m_j).$$

In case (2), message m_i must arrive at j before m by the FIFO channel assumption. Upon receiving m_i, hence before receiving m, j must immediately send out its algorithm messages and reach its *Cut* state. This contradicts the assumption that m was received before the cut. ∎

Theorem 4 *There is an inhibitory consistent-cut protocol for both FIFO and non-FIFO systems using* $3(n-1)$ *messages, where* n *is the number of nodes in the network.*

Our inhibitory protocol is a variant of the classic two-phase commit [8], where a spanning tree of the network is used to minimize communication. It is inhibitory because it disables send events between the two phases of messages. It does not require use of atomic receive-send or atomic multiple send mechanisms. Again there is a distinguished initiator I. Three messages, $PrepareCut$,Cut, and $Resume$ are sent respectively up, down, and back up the tree. System send events are disabled as Cut moves down the tree and re-enabled as $Resume$ moves up the tree.

Algorithm 2 *Inhibitory Consistent-Cut Protocol*

1. Let T be a spanning tree of the network rooted at I. At each process i, let variable $parent(i)$ and list $children(i)$ contain the parent and children of i in T. An *internal node* has both a parent and children; a *leaf node* has only a parent. The *root* node has no parent.

2. Initiator I: Send $PrepareCut$ to all children(I).

3. Each internal node i, after receiving $PrepareCut$: Send $PrepareCut$ to all $children(i)$.

4. Leaf nodes i, after receiving $PrepareCut$: Disable system send events; Send Cut to $parent(i)$. {Cut}

5. Each internal node i, after receving Cut from all $children(i)$: Disable system send events; Send Cut to $parent(i)$. {Cut}

6. Initiator I, after receiving Cut from all $children(i)$: {Cut} Send $Resume$ to all $children(I)$.

7. Internal nodes i, after receiving $Resume$ from $parent(i)$: Send $Resume$ to all $children(i)$; Enable previously disabled system send events.

8. Leaf nodes i, after receiving $Resume$ from $parent(i)$: Enable previously disabled system send events.

Proof of Theorem 4: The local states comprising the cut are: (1) in the initiator, immediately after receiving *Cut* from all of its children, and (2) in an internal or leaf node, immediately after sending *Cut* to its parent.

Suppose message m is sent after the cut by i and is received before it by j. There are three cases to consider: (1) i is the initiator, (2) j is the initiator, and (3) neither are the initiator.

In case (1), illustrated in figure 5(a), let $m_1, ...m_k$ be the sequence of *Cut* messages from j to the initiator. "$\times$" denotes the cut states which immediately follow the sending of each m_i and the initiator's last reception of a *Cut* message from its children (not necessarily immediately after the reception of m_k). Since m is received by j before the cut, it is received before $send(m_1)$. Clearly $send(m_1)$ happens-before $receive(m_k)$. But $receive(m_k)$ is received by the initiator before the cut and hence before sending m. So m would have to be received before sent.

In case (2), illustrated in figure 5(b), the send by i cannot occur until system sends are enabled, which happens after *Resume* is received. Let $m_1, ...m_k$ be the sequence of *Resume* messages from the intiator to i, so $receive(m_k)$ happens-before $send(m)$. If m is received before the cut then it is received before I sends m_1, therefore again the reception must occur before the send.

Case (3) is essentially a concatenation of case (1) and case (2). After the reception of m, i must begins a sequence of *Cut* messages to the initiator who then begins a sequence of *Resume* messages to j. Message m cannot be sent by j until after the resume, with the same result.

Since a spanning tree of the network contains exactly $n - 1$ links and three messages are sent along each of them, the algorithm uses exactly $3(n-1)$ messages. Finally, the fact that channels are FIFO never occurs in the proof of correctness, hence this algorithm works for non-FIFO systems as well. ∎

Theorems 2, 3, and 4 illustrate a tradeoff in many FIFO systems between inhibiting actions of an underlying system and requiring more communication. In particular, for completely connected networks the inhibitory algorithm 2 requires only $3(n-1)$ messages whereas the non-inhibitory algorithm 1 requires up to $n^2 - n$ messages. Indeed, by Theorem 2, no non-inhibitory protocol can do better than $\frac{1}{2}(n^2 - n)$ messages. Also, theorems 1 and 4 illustrate that inhibition is exactly what is required for

(a)

(b)

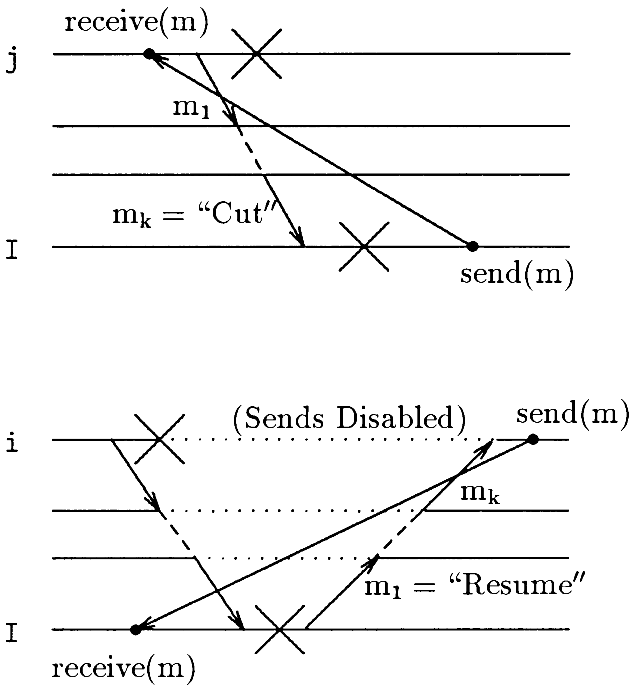

Figure 5: Proof of Theorem 4: (a) case 1, (b) case 2.

consistent-cut protocols in non-FIFO systems corresponding to our model. This requirement generalizes to weaker models without atomic receive-send and atomic multiple send primitives, since those primitives are not necessary in algorithm 2.

6 Conclusions

In this paper we have given formal definitions for *non-inhibitory* protocols and *consistent-cut* protocols. We have shown that there is no non-inhibitory consistent-cut protocol for non-FIFO systems, and given a lower bound on the communication necessary for such a protocol in a FIFO system. In the future we would like to also give a lower bound for inhibitory consistent-cut protocols.

We have presented two consistent-cut protocols; a non-inhibitory FIFO protocol, requiring up to two messages between each pair of neighboring nodes, and an inhibitory one, requiring a total of $3(n-1)$ messages where n is the number of nodes in the network. This illustrates a tradeoff between communication complexity and inhibition of underlying system activity for most FIFO networks. In addition, the inhibitory protocol also works for non-FIFO systems, thus illustrating that inhibition is exactly the property required to achieve a consistent-cut in non-FIFO systems that correspond to our model.

In related work, Awerbuch [1] observes tradeoffs between communication complexity and execution time in asynchronous "synchronizers" similar to algorithms 1 and 2. Whereas our consistent cuts include all send events for which there is a corresponding receive in the cut, the synchronizers generate sets of "pulses," one per process, such that all messages sent before the pulses have been received before them. Thus the problems are symmetric and incomparable in the sense that the former allows sends without receives in cuts and the later allows receives without sends before a set of pulses. More importantly, Awerbuch does not consider the issue of inhibition whereas we do not consider the execution time of algorithms.

Our model includes both indivisible multiple-send events and indivisible receive-send events. Of course, our impossibility and lower-bounds results will still hold in weaker models which do not include these primitives. Additionally, our inhibitory protocol does not require such indivisible events and therefore generalizes to the weaker model. However, it has yet to be proven whether or not these primitives are indeed necessary to develop a non-inhibitory protocol for the FIFO systems we are modelling. This is also a topic under investigation.

7 Acknowledgements

I would like to thank Prakash Panangaden for many helpful discussions throughout the development of this work and Willy Zwaenepoel for critical observations leading to the definition of non-inhibitory and to algorithm 2, as well as Ken Birman, Ajei Gopal, and Amitabh Shah for commenting on earlier drafts of this paper.

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

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

[5] 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.

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

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

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

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