The Fault Span of Crash Failures

GEORGE VARGHESE AND MAHESH JAYARAM

Washington University, St. Louis, Missouri

Abstract. A *crashing* network protocol is an asynchronous protocol whose memory does not survive crashes. We show that a crashing network protocol that works over unreliable links can be driven to arbitrary global states, where each node is in a state reached in some (possibly different) execution, and each link has an arbitrary mixture of packets sent in (possibly different) executions. Our theorem considerably generalizes an earlier result, due to Fekete et al., which states that there is no correct crashing Data Link Protocol. For example, we prove that there is no correct crashing protocol for token passing and for many other resource allocation protocols such as k-exclusion, and the drinking and dining philosophers problems. We further characterize the reachable states caused by crash failures using reliable non-FIFO and reliable FIFO links. We show that with reliable non-FIFO links any acyclic subset of nodes and links can be driven to arbitrary states. We show that with reliable FIFO links, only nodes can be driven to arbitrary states. Overall, we show a *strict* hierarchy in terms of the set of states reachable by crash failures in the three link models.

Categories and Subject Descriptors: C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks; C.2.6 [**Computer-Communication Networks**]: Internetworking; D.4.4 [**Operating Systems**]: Communications Management; D.4.5 [**Operating Systems**]: Reliability; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation

General Terms: Theory

1. *Introduction*

We consider asynchronous network protocols that work with faulty components: links that can lose and permute packets, and nodes that can crash and restart. Many network protocols that are commonly deployed (e.g., HDLC, IP, the OSI and DECNET Routing protocols [Tannenbaum 1996]) come under this cate-

Extended abstracts of the results in this paper appeared in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96).* ACM, New York, 1996, pp. 247–256 and *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97).* ACM, New York, 1997, pp. 179–188.

G. Varghese was supported by National Science Foundation (NSF) grant NCR 94-05444 and an ONR Young Investigator Award.

gory.[1] Crash failures, where a node crashes in the middle of a protocol, are a common cause of protocol failures.

Many existing protocol specifications do not require the nodes executing the protocol to have nonvolatile memory. Thus, if a node crashes and restarts, it can lose all memory of its previous state in the execution. Instead, after the restart, a node comes up in an initial state in which all protocol variables are set to prespecified initial values. This may seem strange to the reader because nonvolatile memory (e.g., disk) appears to be quite cheap and provides useful protection against crash failures.

However, many early protocol implementations were on stand-alone devices (e.g., bridges, routers) that did not have a disk. Adding a disk was precluded by the expense, and sometimes by the physical configuration (e.g., internal buses) of the device. Thus many network protocols like IP and HDLC *do not require* that nodes have nonvolatile memory (NVM). Recently, cheaper electronic forms of NVM (e.g., NVRAM) have become available. However, even these have problems. Some require a battery that may fail; others wear out after being written, say, 10,000 times. Many existing router products use NVRAM only to store management parameters and not to store all protocol state variables. Thus, results about protocols that do not use NVM are interesting, because many existing protocols are in this category.

One sensible way to restart is to rely on bounds on message lifetimes, say, T. If a node waits for some multiple of T after restarting, it can avoid getting confused by responses to pre-crash messages. Such an approach was used in the ARPANET routing protocol [McQuillan et al. 1980] and in the first timer-based transport protocol [Watson 1981]. A disadvantage is that message lifetimes in a large network are high, leading to a noticeable rebooting delay. Thus, in this paper, we assume an *asynchronous* model in which there are no time bounds on message delay or node computation. This is reasonable when time bounds are either too high or too risky to use.

A second possible way to restart is to choose, after a crash, random incarnation numbers that, with high probability, are not present in the network. This approach was advocated for DEC's NSP transport protocol [Digital Equipment Corporation 1983]. However, choosing truly random numbers after a crash is a somewhat delicate problem.[2]

An earlier result by Baratz and Segall [1988] showed that the widely deployed Data Link protocol HDLC could work incorrectly if nodes did not keep nonvolatile memory and the links could lose messages. Later, Fekete et al. [1993] showed that no Data Link protocol could work correctly under these assumptions. Attiya et al. [1995] proved a similar result regarding Transport protocols. In this paper, we investigate the power of crash failures for protocols other than Data Link and Transport protocols. Our first theorem generalizes the Data Link result in Fekete et al. [1993] to a statement that applies to arbitrary protocols. Thus, our theorem can also be used to show new results besides the Data Link

[1] Some routing protocols depend on time bounds and thus are not strictly asynchronous. However, there are subcomponents of these protocols that do not depend on time bounds for their correctness, and hence can be considered to be asynchronous.

[2] Choosing a pseudorandom number based on the node address causes repeated numbers; we can use the clock value as a seed only if the clock survives the crash.

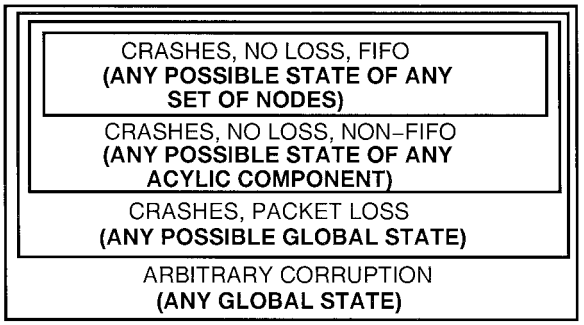

FIG. 1. Summary of fault span results in this paper.

result: For instance, the impossibility of token passing or resource allocation with crash failures and no NVM.

From a theoretical viewpoint, our paper essentially characterizes the *fault span* of crash failures in a particular network model. The fault span is the set of global states that a set of faults can drive a system into. The fault-span defines the power of crash failures—the larger the fault-span, the more dangerous the effect of crash failures. Our results indicate that the fault-span of crash failures (in a network model that applies to many practical settings) is very large. Knowledge of the fault-span can help a protocol designer: the designer must be prepared to deal with all possible states in the fault-span. In particular, if the fault-span includes all combinations of node and link states, the protocol must essentially be self-stabilizing. We show that this is indeed the case for the combination of crash failures and links that can lose messages (unreliable FIFO links). Such links are a good model of many real physical links.

In order to test the sensitivity of our result to the link model used, we go further and investigate the fault span with two other common link models: reliable FIFO and reliable non-FIFO. In both models, the link will not lose messages; however, reliable non-FIFO links can permute messages. Reliable link models are appropriate when the probability of message loss is small compared to that of node crashes, and non-FIFO links are a good abstraction of links that model networks in which there are multiple paths, which allow packets to be received out-of-order.

We show that the fault span for these two models is very different from that of the CAML model, and that the three models fall into a natural complexity hierarchy in terms of the "power" of faults in each model. This is illustrated in Figure 1. Note that the figure does not separate the model for crash failures with unreliable FIFO links from the model for crash failures with unreliable non-FIFO links. This is because we will show that the fault span for these two models is identical.

The rest of this paper is organized as follows: We describe our results intuitively in the next section. Our formal treatment begins with a model in Section 3, some useful notation in Section 4, and continues with a formal statement of the main results in Section 5. Before we prove these results, we describe applications of these results in Section 6. We begin our proofs by introducing notation for send and receive sequences in Section 7. We prove that

crash failures and unreliable links can drive protocols to arbitrary states in Section 8.

Next, we prove that crash failures and reliable FIFO links can drive the nodes to arbitrary states in Section 9. We then prove that crash failures and reliable non-FIFO links can drive any acyclic component of a protocol to arbitrary states in Section 10. We describe counterexample protocols in Section 11 to show that our fault span hierarchy is strict. We show how to design correct crash resilient protocols in Section 12. We state our conclusions in Section 13.

2. *Intuition behind Main Theorems*

In what follows, we will not distinguish between crashes and restarts. What we call a crash can be imagined to be a crash that is immediately followed by restart. Intuitively, a *crashing protocol* is a protocol in which the nodes have no nonvolatile memory (NVM); thus after a crash event, a node goes to a prespecified initial state.[3] Formal definitions are given later in the context of the I/O Automaton model.

In this section, we describe the intuition behind the main results. We start by comparing the power of three link models used and providing an intuitive statement of the three results. We then sketch an important construction underlying the Data Link impossibility result of Fekete et al. [1993], and show how to generalize the construction to a new construction that we call *concatenation*. We then describe intuitively how concatenation can be used to present the results for all three link models. We note that the formal proofs for general graphs that we present later are much more complicated than the simple two node sketches we present below. However, once the basic intuition is grasped, the formal proofs are easier to follow.

2.1. Naming and Comparing Link Models.

We will use *CAML model* (for Crashing, Asynchronous, Memoryless, and Lossy) to denote the combination of a asynchronous protocol subject to crash failures, that has no access to NVRAM, and works over unreliable FIFO links. We will use *CAMO model* to denote a similar combination (the *O* stands for out-of-order) except that we substitute reliable non-FIFO links for unreliable FIFO links. Finally, we use *CAM model* to denote a similar system using reliable FIFO links.

We can understand the relative power of the different models more clearly by looking at an example. Suppose we have a two-node protocol and the link from say Node S to say Node R contains the sequence of packets $p_1\, p_2\, p_3$ such that p_1 is at the head of the link.[4] Suppose we would like to remove packet p_2 and leave the rest of the state unchanged as far as possible.

In the CAML model, this is easily done by losing packet p_2. In the CAMO model, as the links are reliable we cannot lose p_2 directly. Since links are non-FIFO, we can first reorder packets so p_2 is at the head of the link, receive p_2 at Node R and then "lose" p_2 by crashing Node R. This does remove p_2 from the link but leaves Node R in its initial state. In the CAM model, as links are reliable

[3] Thus we model as a single event the crash and the subsequent clean-up and restart; since no protocol activity occurs until restart is over, there is no loss of generality.

[4] The convention used throughout the paper is that whenever a sequence of packets on a link is written, the leftmost packet in the sequence denotes the packet at the head of the link.

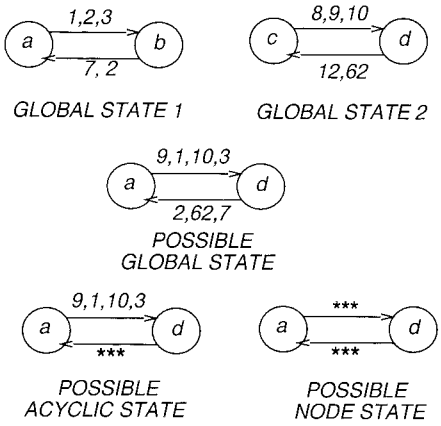

FIG. 2. Possible global, acyclic, and node states constructed from two different global states. These correspond to the reachable global states of the CAML, CAMO, and CAM models, respectively.

and FIFO, the only way to remove p_2 is to receive p_1 as well as p_2, and then to lose the packets by crashing Node R. Thus, we see that the three models CAML, CAMO, and CAM have progressively decreasing power.

We note that our results in the CAMO and CAM models depend on the ability to crash a node after it receives a packet but before it sends any further packets. Some models of distributed computation allow a node to receive a packet and send packets in a single atomic action, which would preclude this ability. In a real system, however, while a packet reception can trigger the sending of other packets, these packets can only be queued to outbound link queues and cannot be immediately sent on the link. A subsequent crash would result in the loss of these packets. Our model of computation will reflect this aspect of real systems by treating packet reception and packet sending as separate atomic events.

2.2. PREVIEW OF RESULTS. Before we state, the results for the three models we first describe some terminology which is useful in describing the results. We stated earlier that crashes and lossy links can drive asynchronous protocols to "essentially arbitrary" global states. We clarify what we mean by "arbitrary" by defining *possible node and link states*.

Define a *possible packet* on a link to be a packet that could have been produced on that link in some finite execution. Also define a *possible node state* of a node to be a state reachable by the node in some finite execution. We now define a *possible global state* to be an assignment of: (a) A possible node state to every node, and (b) an arbitrary sequence of possible packets for every link.

Possible states are not more restrictive than truly arbitrary protocol states because we can always modify a protocol to get rid of unreachable node states and to ignore invalid packets. Such checks do not prevent arbitrary *combinations* of possible node states and possible packets on links, where each node state and packet can be drawn from a different global state. This is shown in Figure 2 where the possible global state is constructed by "cutting and pasting" from Global States 1 and 2 shown above. Note that the state of the leftmost node is drawn from Global State 1, the state of the rightmost node from Global State 2, and the packets on links are an arbitrary permutation of packets drawn from the corresponding links in Global States 1 and 2.

For the CAML model, our result, stripped of the formal framework, essentially states: *any crashing protocol that works in the CAML model can be driven into any possible global state*. Thus, the CAML model is essentially equivalent to arbitrary memory corruption at nodes and arbitrary packets on links.

Before we state the results for the CAMO model we define the notion of a *possible global acyclic state*.[5] Consider a network of protocol nodes and communication links. Given an acyclic subset of links and nodes within the network, we define a *possible global acyclic state* to be an assignment of a possible node state to each node in the acyclic subset and an arbitrary sequence of possible packets to each link in the acyclic subset. The result for the CAMO model states: *Any crashing protocol that works in the CAMO model can be driven into any possible global acyclic state*. Thus, given any subset of links and nodes within the network such that there is no cycle within them, we can drive the state of each node and link in that subset to any possible state. However, we cannot control the state of the remaining nodes and links that are not in the subset. Thus, clearly the fault span of the CAML model is greater than that of the CAMO model. This is shown in Figure 2 where the possible acyclic state can only specify possible states for two nodes and one link; the state of the bottom link is shown as *** to indicate that this link cannot be controlled.

To characterize the fault span of the CAM model, we define a *possible global node state* to be an assignment of a possible node state to each node. The result for the CAM model states: *Any crashing protocol that works in the CAM model can be driven into any possible global node state*. The state of the links cannot be controlled. Note that the fault span of the CAMO model is clearly greater than that of the CAM model. This is shown in Figure 2.

Thus, for a two-node node protocol with two unidirectional links connecting the two nodes: In the CAML model, we can drive both nodes and both links to arbitrary states; in the CAMO model, we can drive both nodes and any *one* link to arbitrary states; in the CAM model, we can drive both nodes to arbitrary states. These results are consistent with Figure 1.

2.3. Concatenation Construction. The Data Link impossibility result of Fekete et al. [1993] that we will call the FLMS result) shows that there is no correct crashing Data Link protocol. In a typical Data Link protocol like HDLC, after a sender crash the sender sends a reset or handshake packet and initializes its state after receiving a handshake response from the receiver. The protocol can fail because a sequence of past crashes can initialize the receiver–sender link with a sequence of "old" packets. This sequence includes an ack that fools the sender into thinking that all its sent messages have arrived, when, in fact, they have not.

Our example construction begins with a construction that underlies the FLMS result. We describe the intuition for a two node protocol only. Consider a two-node protocol with a pair of unidirectional links between the nodes. Fix a link L. A send sequence on link L is a sequence of packets that was sent on link L in some finite execution. For example, Figure 3 shows an example execution of a two-node protocol and a send sequence on the receiver–sender link. Notice

[5] When clear from the context, we will sometimes drop the term "global" when referring to possible global states.

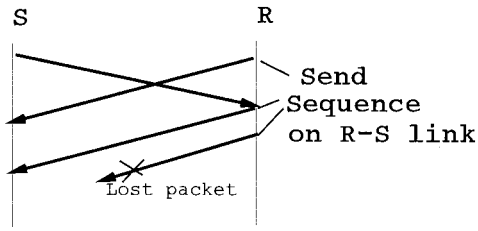

FIG. 3. An example of a send sequence on a link.

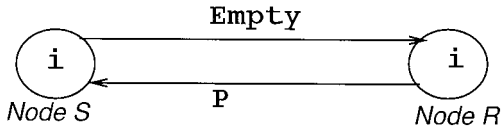

FIG. 4. What the FLMS construction produces. The i at nodes represents the initial state of the corresponding node. P is a send sequence for the link.

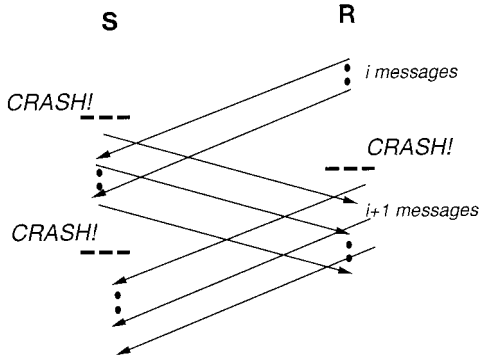

FIG. 5. The essence of the FLMS construction to initialize a link with a sequence of old packets from a past incarnation.

that this sequence contains *all* packets sent on link L from the start of the execution.

Essentially, the FLMS construction shows that in the CAML model, one can find a series of crashes that leave a two node protocol in a state where all nodes are in initial states (i.e., in states that nodes revert to after a crash), and all links are empty except for a single link that has the entire sequence sent in a particular execution. This is illustrated in Figure 4.

The essence of the basic FLMS construction is shown in Figure 5. A series of alternating crashes are used to force a node R to send the first i packets of the sequence of normal crashless packets. This causes the other node S (after another crash) to send the packets needed to force node R to send the first $i + 1$ packets before crashing again. By continuing inductively, we force the receiver to emit the entire sequence of packets it would have emitted in an execution. At this point we stop the construction, and crash the sender and receiver. The result is that the link from the receiver to sender has the complete sequence of packets sent in an execution. (This is sufficient to cause Data Link protocols to fail because the complete sequence could include the responses to any initial handshake packets, as well as all the data acks. Thus, even if all the sender's

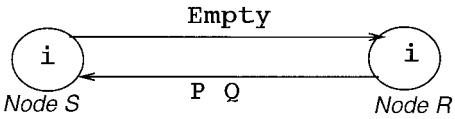

FIG. 6. Our main construction is one that produces a concatenation of two send sequences. Compare with Figure 4.

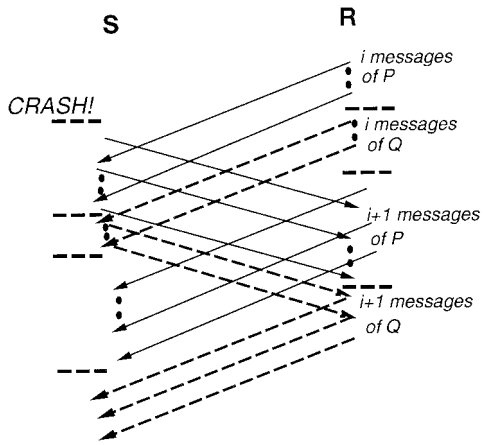

FIG. 7. How the concatenation construction works. The horizontal dashed lines represent crashes. Notice how the construction dovetails two independent FLMS constructions for P and Q.

initial packets are lost, including the first data item, the sender will be fooled into thinking all is well.)

Our first step is to generalize the FLMS construction to show that we can initialize a link with a *concatenation* of two send sequences P and Q, where P and Q are send sequences from two possibly different executions of the same crashing protocol. The construction is depicted in Figure 6.

For two nodes, we can construct the concatenation, say, PQ, in the same way as the FLMS construction, except that we construct P and Q at the same time. Recall that P is constructed inductively by having the first i packets in P being produced, which causes S to send a sequence that in turn causes R to produce the first $i + 1$ packets of P, and so on. Assume that at the same time as we produce the first i packets of P, we crash R and produce the first i packets of Q. Then, as before, we crash S, receive the first i packets of P, and cause S to emit the packets required for R to produce the first $i + 1$ packets of P. But in the new construction we crash S again, cause it to receive the first i packets of Q and emit the packets required for R to produce the first $i + 1$ packets of Q. Thus, two independent FLMS constructions are dovetailed. This is illustrated in Figure 7 using solid lines for the packets of P and dashed lines for the packets of Q. Contrast this figure to the FLMS construction in Figure 5.

There are some subtleties to the construction that we have glossed over. First, if the two sequences have different lengths, at some point we must stop extending one of the two sequences and keep increasing the longer sequence. Second, the construction does not generalize well to multiple nodes and different topologies. In our formal development, we will use a different construction (that essentially

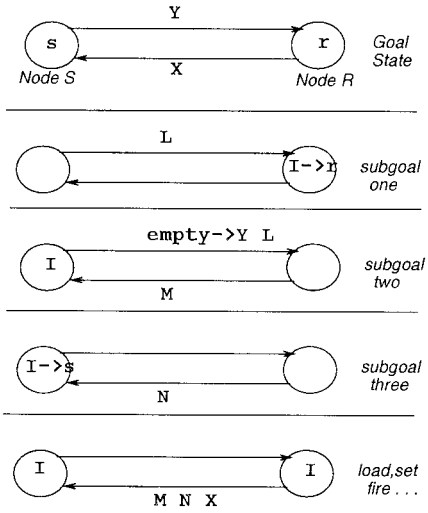

FIG. 8. Driving a CAML system into any possible state by loading one link with the appropriate sequence of packets and by playing out these packets.

constructs the send sequence for *all* links of an execution at the same time) for concatenation.

Notice that the construction so far does not depend on whether the link can lose or permute packets. For non-FIFO links the definition of the concatenation of two sequences is simply the union of the two sequences since order is irrelevant for a non-FIFO link.

By doing the concatenation construction repeatedly, we can concatenate an arbitrary number of send sequences. With this ability, we can produce an arbitrary sequence of possible packets. This is because, by definition, any possible packet occurs in some send packet sequence. Thus, we first concatenate the required number of send packet sequences such that the sequence we require is a subsequence. Then we use the fact that we can lose packets to get the required sequence. Losing can be done directly in the CAML, but must be done indirectly in the CAMO model by delivering the packet to be dropped and then crashing the receiving node. Losing an arbitrary packet p is not possible for the CAM model without losing all the packets ahead of p on the link.

We now proceed to intuitively describe how to use concatenation and loss to derive the desired results for the CAML, CAMO, and CAM models. We show the results only for the two node case; the construction for general graphs is more intricate and described later in the course of the formal proofs. Note that similar mechanisms are used in reaching the goal state in all three models—for example, concatenation and playout of the concatenated packets. The formal proofs also reflect this similarity.

2.4. CONSTRUCTING ANY POSSIBLE STATE IN THE CAML MODEL. Once we can construct an arbitrary sequence of possible packets on a link, we can essentially drive a CAML system into any *possible* global state. This is illustrated in Figure 8. Suppose we want to reach the goal state shown in the top frame. Consider the subgoal of driving node R into state r from an initial state. Since r is a possible state, there must be some sequence of packets L that can drive the receiver from an initial state to state r. That leads directly to a second subgoal of finding a sequence of packets M (on the reverse link this time) that can cause

node S to emit the sequence L as well as the sequence Y we need for the goal state.

Generating L is easy because L is a send sequence; thus, there must be some sequence of packets that can drive node S to emit L. Getting the Y is slightly more tricky. But we observe that any Y is a subsequence of some concatenation C of send sequences. By generating each such send sequence and crashing node S in between each generation, we get C, and finally obtain Y by losing packets. Similarly, we have a third subgoal (similar to the first subgoal) to drive node S to state s using some sequence N.

We finish (see last frame in Figure 8) the construction by "loading" the reverse link with the sequence $M\ N\ X$ (where M is at the head of the link). We now playout some of these packets to achieve the desired goal. We first allow node S to receive M and emit C and L. By losing the appropriate packets in C, we are left with $Y\ L$ on the S to R link, with L at the head of the link (see subgoal 2). Then we crash node S and allow it to receive N and go to state s. Any packets emitted by node S are lost. Finally, we allow node R to receive L (thereby leaving Y on the forward link) and go to state r (see subgoal 1). Any packets sent by node R are lost. This leaves the system in the goal state.

2.5. Driving CAMO to Any Acyclic State. We show how to drive a protocol in the CAMO model to any acyclic state. We assume the concatenation construction.

Suppose we want to reach the goal state shown in the top frame of Figure 9. Thus, we want to control the state of all components except the link from R to S. Consider the subgoal of driving node R into state r from an initial state. Since r is a possible state, there must be some sequence of packets L that can drive the receiver from an initial state to state r. While going to state r, node R emits the sequence of packets O. Consider a second subgoal of finding a sequence of packets M (on the reverse link) that can cause Node S to emit the sequence L followed by the sequence Y we need for the goal state.

L is a send sequence, so there must be some send sequence of packets that can drive node S to emit L. As in the case for CAML, Y is a subsequence of some concatenation C of send sequences. By generating each such send sequence and crashing node S in between each generation, we get C and finally obtain Y by losing the extra packets at node R (as links are reliable in CAMO, so packets must be lost at the receiving end of a link). Similarly, we have a third subgoal (similar to the first subgoal) to drive node S to state s with some sequence N, while allowing S to emit sequence O'.

As before, we use the concatenation construction to load the link from R to S with $M\ N$. We then first allow node S to receive M and emit $L\ Y$ (along with the extra packets in C). Then we lose the extra packets in C to get $L\ Y$. Then, we play out the sequence N that drives S to its final state s while emitting the sequence O'. O' is then moved to the front of the non-FIFO link and dropped at R. Then L is played out, leaving the link with Y and driving the node S to state r. Thus, we arrive at the final goal state. Note that we cannot control the state of the link from R to S, which will contain the sequence of packets O sent by node R. Thus, we have informally shown how to drive a two node protocol in the CAMO model to any acyclic state.

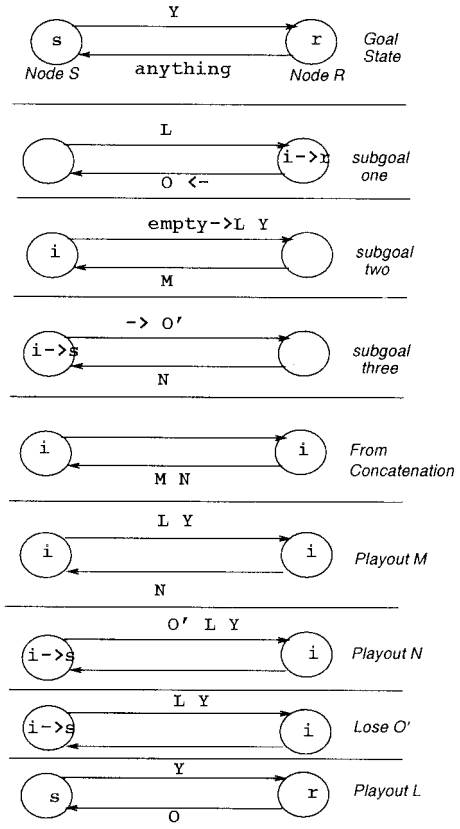

FIG. 9. Driving CAMO to any acyclic state.

2.6. DRIVING CAM TO ANY NODE STATE. In this section, we discuss the result for the CAM model in the same way as we did for the CAMO model. The result for the CAM model states that the system can be driven to a state in which the nodes have any possible state but the links cannot be controlled.

Suppose we want to reach the goal state shown in the top frame of Figure 10 in which the nodes are at some possible states s and r. Consider the subgoal of driving node R into state r from an initial state. Since r is a possible state, there must be some sequence of packets L that can drive the receiver from an initial state to state r. While going to state r, node R emits the sequence of packets O. Consider a second subgoal of finding a sequence of packets M (on the reverse link) that can cause node S to emit the sequence L. Actually, node S on receiving M will emit a sequence L' of which L is a prefix. But as we are not concerned with the state of the links, the remainder of L' can remain on the link in the goal state. The reasoning for the existence of some such M is similar to the reasoning for the other models. Similarly, we have a third subgoal (similar to the first subgoal) to drive node S to state s with some sequence N, while allowing S to emit sequence O'.

As before, we use the concatenation construction to load the link from R to S with M N. We then first allow node S to receive M and emit L (along with the extra packets in L'). Then we play out the sequence L which is the prefix of L' to drive R to state r, while emitting the sequence O. Then the sequence N is

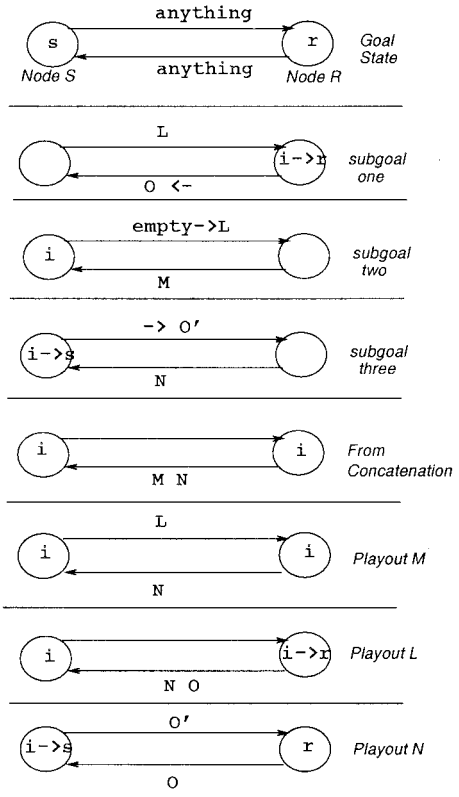

FIG. 10. Driving CAM to any node state.

played out to S to drive S to final state s, while emitting the sequence O'. Thus, we reach the goal state.

Note that we cannot control the state of the two links that have the sequence of packets O and O' respectively in the final state. Thus, we have informally shown how to drive a two-node protocol in the CAM model to any node state.

3. *Model*

We use the Input/Output Automaton model of Lynch and Tuttle [1989] for modeling protocols. This model is essentially a state machine model that allows us to compose state machines representing links and nodes. We review the essential notation here, but refer the reader to Lynch and Tuttle [1989] for more details.

3.1. INPUT OUTPUT AUTOMATA (IOA).

Informally, an IOA is a state machine whose state transitions are given labels called *actions*. There are three kinds of actions. The environment affects the automaton through *Input actions* that must be responded to in any state. The automaton affects the environment through *Output actions*; these actions can be controlled by the automaton to only occur in certain states. *Internal actions* only change the state of the automaton without affecting the environment.

Formally, an IOA is an automaton defined by a *state set* S, a *action set* A, an *action signature* Z (that classifies the action set into input, output, and internal

actions), a *transition relation* $R \subseteq S \times A \times S$, and a set of *initial states* $I \subseteq S$. An action a is said to be *enabled* in state s if there exist $s' \in S$ such that $(s, a, s') \in R$. Input actions are always enabled.

When an IOA "runs" it produces an execution. An *execution fragment* is an alternating sequence of states and actions $(s_0, a_1, s_1, \ldots)$, such that $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. An execution fragment E is fair if any internal or output action that is continuously enabled eventually occurs. The IOA model actually specifies fairness in terms of equivalence classes and the definition really applies to all actions in a class. For our purposes in this paper, we can consider each internal and output action as being in a separate fairness class. Thus, unlike conventional IOA descriptions, we will not spell out the fairness classes for any IOA we describe. Finally, an *execution* is an execution fragment that begins with a start state.

There is a notion of *composition* [Lynch and Tuttle 1989] that produces a composite automaton out of constituent automata. Input and output actions of the same name are performed simultaneously. Thus, when a node automaton i performs a $send^{i,j}(p)$ output action, if the link automaton between i and j has a input action of the same name, then the link performs the corresponding input action (typically to store p). The state of the composite automata is the composition of the states of the constituent automata.

The *schedule* of an execution fragment α of A is the subsequence of α consisting of all the actions with the states removed, and is denoted by $sched(\alpha)$. We say that β is a *schedule* of A if β is the schedule of an execution of A. The *behavior* of an execution fragment α of A is the subsequence of α consisting of only the external (i.e., input and output) actions with the states and internal actions removed. We say that β is a *behavior* of A if β is the behavior of an execution of A. We define a *fair schedule* and a *fair behavior* analogously.

The following lemma states that it is always possible to extend a finite execution of any automaton to a fair execution of the automaton.

LEMMA 3.1 (FAIR EXTENSION). *If α is a finite execution of an automaton A, then there exists an extension α' of α that contains no more input actions and is a fair execution of A.*

3.2. MODELING CRASHING NETWORK PROTOCOLS. We model a protocol as a composition of automata, one for each node representing the protocol agent at that node, and one for each pair of neighboring nodes representing the unidirectional communication link between the nodes. We model the network topology using a directed graph $G = (V, E)$ where $n = |V|$. Nodes of a protocol communicate by sending and receiving *packets*. Fix a packet alphabet P. A *protocol* for graph $G = (V, E)$ is a tuple $A = (A^1, A^2, \ldots, A^n)$ of *node automata* A^i for each $i \in V$. Each node automaton A^i has output actions $send^{i,j}(p)$, $p \in P$ (to send packets to neighbor j) for each j such that $(i, j) \in E$, and input actions $receive^{j,i}(p)$, $p \in P$ (to receive packets from neighbor j) for each j such that $(j, i) \in E$.

A *crashing automaton* is an automaton X that has an input action, say, *crash*, such that if s_0 is the unique start state of X, then for all states s of X, $(s, crash, s_0)$ is a transition of X. A *crashing protocol for a graph* $G = (V, E)$ is a tuple $A = (A^1, A^2, \ldots, A^n)$ of crashing node automata A^i for each $i \in V$, where the crash action for node i is called $crash^i$.

Input actions:
  $send^{i,j}(p)$
  Effect: $count^{i,j}[p] \leftarrow count^{i,j}[p] + 1$
    append $(p, count^{i,j}[p])$ to tail of $queue^{i,j}$.
Output actions:
  $receive^{i,j}(p)$
  Precondition: (p, k) is at the head of $queue^{i,j}$.
  Effect: remove (p, k) from head of $queue^{i,j}$.
Internal actions:
  $lose^{i,j}(p, k)$
  Precondition: $(p, k) \in queue$ and $k \notin keep^{i,j}[p]$.
  Effect: remove (p, k) from $queue^{i,j}$.

FIG. 11. Unreliable FIFO link automaton.

3.3. MODELING FIFO UNRELIABLE LINKS. Any unreliable FIFO link model must satisfy the following reasonable properties: only packets that are sent are received, the link obeys the FIFO property, and the link is live. For liveness, we require that a packet that is sent an infinite number of times is received an infinite number of times. Let us call any sequence of *send* and *receive* actions which satisfies the above properties *U-consistent*. Let us call a link automaton *universal* if its fair behaviors are all the sequences of actions which are *U-consistent*.

Fekete et al. [1993] describe a universal link automaton, which we call *U-universal*^{i,j}. We now describe our universal link automaton, $\mathrm{U}^{i,j}$, which models an unreliable FIFO link between nodes i and j.

Note that an unreliable data link without a liveness guarantees can easily be modeled by just using a queue of packets and an action to lose packets. However, extra complexity is needed to model a Data Link that will ensure liveness properties, despite the possibility of losing packets. We use a small variation of the link model in Fekete et al. [1993], which is proved in Fekete et al. [1993] to be a universal link automaton. We prove in Jayaram [1996] that our link automaton $\mathrm{U}^{i,j}$ is equivalent to the universal link automaton of Fekete et al. [1993]. We prefer our variation because it isolates packet loss in a separate action, which is more convenient for our proofs.

$\mathrm{U}^{i,j}$ has an input action $send^{i,j}(p)$ by which node i sends a packet to node j. It has a $receive^{i,j}(p)$ output action by which node j receives packet p. It also has an internal $lose^{i,j}$ action for losing packets. The state of $\mathrm{U}^{i,j}$ consists of a queue (i.e., a sequence), $queue^{i,j}$, each element of which is a pair (p, k) where p is a packet and k is an integer, an array $count^{i,j}$ of integers indexed by packet values, and an array $keep^{i,j}$ of infinite sets of positive integers indexed by packet values. The queue contains packets as well as the counts at which the packets were sent. The second component is used by the *lose* action to identify packets to lose.

The initial states of the automaton are those states in which $queue^{i,j}$ is empty and each entry $count^{i,j}[p]$ is zero. Thus each initial state is determined by $keep^{i,j}$. The actions of $\mathrm{U}^{i,j}$ are shown in Figure 11.

Intuitively, for each packet p, $keep^{i,j}[p]$ contains an infinite set of positive integers that represent sending attempt numbers for sending p that are guaranteed to succeed. In order to enforce this, $count^{i,j}[p]$, counts the number of attempts so far to send p. We tag every packet with its current attempt number before placing it in the queue, and do not allow the packet to be lost if its

attempt number is in the *keep* set for that packet. This ensures that an infinite number of attempts to send p will result in an infinite number of deliveries of p. Note that this is only a *model*. A real link will mimic this behavior by other means, such as losing packets with a small loss probability.

The reader who wishes to can safely skip the details of how the $count^{i,j}$ array is used to ensure liveness and think of the state of the link automaton as only the sequence consisting of packets in $queue^{i,j}$. This is because our proofs use *finite* constructions that do not rely on the fairness properties of universal links. We only provide the live Data Link specification for completeness; clearly one would need the liveness properties to prove that a protocol is correct, though they are not needed to show that a protocol is incorrect.

Recall that in our version of the IOA model, a fair execution is one in which every continuously enabled internal or output action eventually occurs. Thus, in the unreliable FIFO automaton, any continuously enabled *lose* or *receive* action must eventually occur. We will sometimes use the fact that every finite execution has a fair extension.

A *crashing automaton* $A(\mathrm{U})$ for graph $G = (V, E)$ is the composition of crashing node automata A^i for all $i \in V$ and $\mathrm{U}^{i,j}$ for all $(i, j) \in E$. This will represent a generic system in the CAML model. Recall that CAML is a shorthand for *C*rashing, *A*synchronous, *M*emoryless, and *L*ossy.

3.4. Reliable non-FIFO Links. A reliable non-FIFO link should satisfy the following properties: only packets that are sent are received, and the link is live—that is, if a packet is sent, it is eventually delivered. Once again, liveness is a problem since a non-FIFO link can deliver packets in any order, and all packets in the link are potentially enabled for delivery. So one must guard against the eventuality of a packet remaining forever in the link while later packets get delivered. Once again, modeling liveness adds some complexity to the model.

We now describe the link automaton, $\mathrm{RN}^{i,j}$, which models a reliable non-FIFO link between nodes i and j. $\mathrm{RN}^{i,j}$ has an input action $send^{i,j}(p)$ by which node i sends a packet to node j. It has a $receive^{i,j}(p)$ output action by which node j receives packet p. The state of $\mathrm{RN}^{i,j}$ consists of a set, $set^{i,j}$, each element of which is a pair (p, k) where p is a packet and k is an integer, an array $count^{i,j}$ of integers indexed by packet values, and $tags^{i,j}$, an infinite two dimensional array of positive integers indexed by packet values and integers. The array $count^{i,j}$ keeps track of sequence numbers of packets and $tags^{i,j}$ is an array just like the array *keep* for unreliable FIFO links, which is used to ensure liveness of the link. A proof of liveness of the reliable non-FIFO link is discussed in Jayaram [1996]. The set contains packets as well as the integers associated with the packets. This integer is used by the *receive* action to identify the packet to deliver.

The initial states of the automaton are those states in which $set^{i,j}$ is empty, and each entry $count^{i,j}[p]$ is zero. Thus, each initial state is determined by the array $tags^{i,j}$, which must satisfy the following constraint: all values in the $tags^{i,j}$ array are unique. Intuitively, $tags^{i,j}$ encodes the sequence in which the packets are supposed to be received in an execution. The actions of $\mathrm{RN}^{i,j}$ are shown in Figure 12.

The liveness of the link follows from the fact that as each tag is finite, given that (p, k) is in $set^{i,j}$, only a finite number of events can elapse before p is received. Also note that by initializing the $tags^{i,j}$ array appropriately, one can get

Input actions:
  $send^{i,j}(p)$
  Effect: $count^{i,j}[p] \leftarrow count^{i,j}[p] + 1$
    add $(p, tags^{i,j}[p][count^{i,j}[p]])$ to $set^{i,j}$.
Output actions:
  $receive^{i,j}(p)$
  Precondition: (p, k) is in $set^{i,j}$ and for all other (p', k') in $set^{i,j}$, $k < k'$.
  Effect: remove (p, k) from $set^{i,j}$.

FIG. 12. Reliable non-FIFO link automaton.

all possible live executions of a reliable non-FIFO link. In Jayaram [1996], we prove the liveness properties in greater detail.

As in the case of the FIFO unreliable link, the reader can think of the link state as only consisting of $set^{i,j}$ because the *tags* and *count* arrays are only used to specify a live non-FIFO link.

A *crashing automaton* $A(\mathrm{RN})$ for graph $G = (V, E)$ is the composition of crashing node automata A^i for all $i \in V$ and $\mathrm{RN}^{i,j}$ for all $(i, j) \in E$. This will represent a generic system in the CAMO model. (Recall that the only difference between the CAML and CAMO models is that in the CAMO model, the link allows *O*ut-of-order delivery but cannot lose packets.)

3.5. RELIABLE FIFO LINKS. A reliable FIFO link is equivalent to a lossless queue of packets that has the following properties: Only packets that are sent are received and the link obeys the FIFO property—that is, packets are received in the order in which they are sent. Liveness is not an issue as the link is reliable and there is no packet loss. We now describe the link automaton, $\mathrm{R}^{i,j}$ which models a reliable FIFO link between nodes i and j.

$\mathrm{R}^{i,j}$ has an input action $send^{i,j}(p)$ by which node i sends a packet to node j. It has a $receive^{i,j}(p)$ output action by which node j receives packet p. The state of $\mathrm{R}^{i,j}$ consists of a queue $queue^{i,j}$, each element of which is a packet p. In the initial state of the automaton $queue^{i,j}$ is empty. The actions of $\mathrm{R}^{i,j}$ are shown in Figure 13.

A *crashing automaton* $A(\mathrm{R})$ for graph $G = (V, E)$ is the composition of crashing node automata A^i for all $i \in V$ and $\mathrm{R}^{i,j}$ for all $(i, j) \in E$. This will represent a generic system in the CAM model.

4. *Vector Notation for Global States*

We use vector notation to succinctly describe and prove our results. The state of the system is expressed using a *node state vector* and a *link state vector*. A node state vector is a vector with a component for each node state. A link state vector is a two-dimensional vector (matrix) that has a component for each link.

Fix a protocol automaton, say with unreliable FIFO links, $A(\mathrm{U})$ for graph $G = (V, E)$. Formally, if $[\mathbf{N}, \mathbf{L}]$ denotes the state s of $A(\mathrm{U})$, then for all $i \in V$, $\mathbf{N}[i] = s|A^i$ and for all $(i, j) \in E$, $\mathbf{L}[i, j]$ = the sequence consisting only of packets in $s|\mathrm{U}^{i,j}$. More formally, if $s|\mathrm{U}^{i,j}.queue^{i,j} = (p_1, k_1)(p_2, k_2) \cdots$, then $\mathbf{L}[i, j] = p_1 p_2 \cdots$. By convention, we assume that the first packet of the sequence (p_1 in the above case) is at the "head" of the link. In Figure 14, we illustrate the notation for a three-node ring. All vectors are written in bold in the rest of the paper.

Input actions:
$send^{i,j}(p)$
Effect: append p to the tail of $queue^{i,j}$.
Output actions:
$receive^{i,j}(p)$
Precondition: p is at the head of $queue^{i,j}$.
Effect: remove p from the head of $queue^{i,j}$.

FIG. 13. Reliable FIFO link automaton.

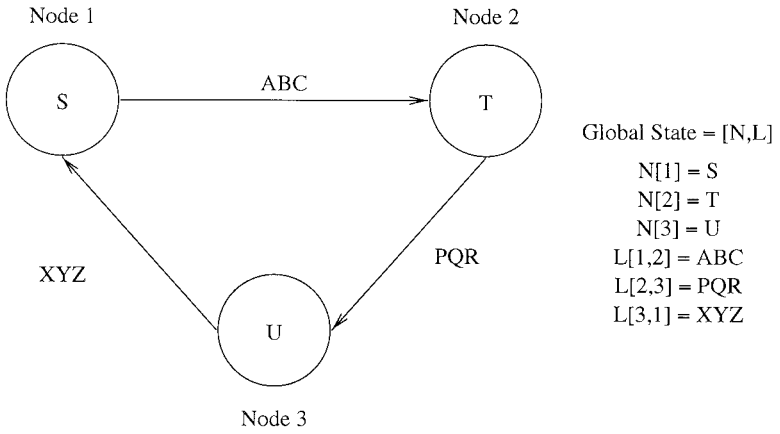

FIG. 14. Vector notation for the state of a three-node ring.

Similarly, consider a protocol automaton with reliable FIFO links, $A(\mathrm{R})$, for graph $G = (V, E)$. If $[\mathbf{N}, \mathbf{L}]$ denotes the state s of $A(\mathrm{R})$, then for all $i \in V$, $\mathbf{N}[i] = s|A^i$ and for all $(i, j) \in E$, $\mathbf{L}[i, j] =$ the sequence consisting only of packets in $s|\mathrm{R}^{i,j}$. More formally, if $s|\mathrm{R}^{i,j}.queue^{i,j} = p_1 p_2 \cdots$, then $\mathbf{L}[i, j] = p_1 p_2 \cdots$.

Finally, consider a protocol with reliable non-FIFO links, $A(\mathrm{RN})$ for graph $G = (V, E)$. If $[\mathbf{N}, \mathbf{L}]$ denotes the state s of $A(\mathrm{RN})$ then for all $i \in V$, $\mathbf{N}[i] = s|A^i$ and for all $(i, j) \in E$, $\mathbf{L}[i, j] =$ the set consisting only of packets in $s|\mathrm{RN}^{i,j}$. More formally, if $s|\mathrm{RN}.set^{i,j} = \{(p_1, k_1), (p_2, k_2), \ldots\}$, then $\mathbf{L}[i, j] = \{p_1, p_2, \ldots\}$.

We now define the concatenation of link state vectors for FIFO links. $\mathbf{K} = \mathbf{L}\ \mathbf{M}$ is the *concatenation of link state vectors* $\mathbf{L}$ and $\mathbf{M}$ if $\mathbf{K}[i, j] =$ the sequence $\mathbf{L}[i, j]\ \mathbf{M}[i, j]$, for all $(i, j) \in E$.

For non-FIFO links, concatenation of link state vectors is defined as follows: $\mathbf{K} = \mathbf{L}\ \mathbf{M}$ is the *concatenation of link state vectors* $\mathbf{L}$ and $\mathbf{M}$ if $\mathbf{K}[i, j] =$ the union $\mathbf{L}[i, j] \cup \mathbf{M}[i, j]$, for all $(i, j) \in E$.

Let $\mathbf{N_0}$ be the node state vector such that for every $i \in V$, $\mathbf{N_0}[i]$ is the unique start state of node A^i. Let $\mathbf{L_0}$ be the link state vector such that for every $i, j \in E$, $\mathbf{L_0}[i, j] = \epsilon$, the empty sequence/set. The start state of $A(\mathrm{U})$, $A(\mathrm{R})$, $A(\mathrm{RN})$ for every execution is $[\mathbf{N_0}, \mathbf{L_0}]$. Given an execution α we use $acts(\alpha)$ to denote the sequence of actions in α. We use $acts^i(\alpha)$ to denote $acts(\alpha)|A^i$ (the subsequence of $acts(\alpha)$ projected on to A_i).

We describe state transitions as follows: We use the notation $[\mathbf{N}, \mathbf{L}] \overset{\beta'}{\rightsquigarrow} [\mathbf{N}', \mathbf{L}']$ to denote that the finite sequence of actions β' takes the system from state $[\mathbf{N}, \mathbf{L}]$

to state $[\mathbf{N}', \mathbf{L}']$. The notation denotes that, if there exists a finite schedule β that takes the system to state $[\mathbf{N}, \mathbf{L}]$, then there exists a finite schedule $\beta\beta'$ that takes the system to state $[\mathbf{N}', \mathbf{L}']$. We will drop the superscript β' and use the notation $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}', \mathbf{L}']$ to denote that there exists *some* finite sequence of actions that takes the system from state $[\mathbf{N}, \mathbf{L}]$ to state $[\mathbf{N}', \mathbf{L}']$. The transition operation $\rightsquigarrow$ is *transitive*: If $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}', \mathbf{L}']$ and $[\mathbf{N}', \mathbf{L}'] \rightsquigarrow [\mathbf{N}'', \mathbf{L}'']$, then $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}'', \mathbf{L}'']$.

4.1. Possible States and Vectors. Given a finite execution $\alpha = s_0\, a_1\, s_1\, a_2\, s_2 \cdots a_n\, s_n$ of $A(\mathrm{U})$ let $last\_state^j(\alpha)$ denote $s_n|A^j$, the state of node A^j in the final state s_n. Let the sequence of packets sent on link $\mathrm{U}^{j,k}$ in any finite execution α of $A(\mathrm{U})$ be $snd^{j,k}(\alpha)$.

Let us define s as a *possible node state* of a node A^j of $A(\mathrm{U})$ if $s = last\_state^j(\alpha)$ for some finite execution α of $A(\mathrm{U})$. A *possible node state vector* of $A(\mathrm{U})$ is a node state vector in which each component is a possible node state.

A *possible packet* on a link $\mathrm{U}^{i,j}$ of $A(\mathrm{U})$ is a packet which is sent on that link in some execution of $A(\mathrm{U})$. More formally, if p is a possible packet on link $\mathrm{U}^{i,j}$ of $A(\mathrm{U})$, then there exists a finite execution α of $A(\mathrm{U})$ such that $p \in snd^{i,j}(\alpha)$. A *possible link state* of a link $\mathrm{U}^{i,j}$ of $A(\mathrm{U})$ is defined as any sequence of possible packets for that link. A *possible link state vector* $\mathbf{L}$ of $A(\mathrm{U})$ is a link state vector such that for all $(i, j) \in E$, $\mathbf{L}[i, j]$ is a possible link state of link $\mathrm{U}^{i,j}$ of $A(\mathrm{U})$. Finally, we define a *possible state* of $A(\mathrm{U})$ as a state $[\mathbf{N}, \mathbf{L}]$ such that $\mathbf{N}$ is a possible node state vector and $\mathbf{L}$ is a possible link state vector.

Though we have defined the above terms for a system with unreliable FIFO links, $A(\mathrm{U})$, they are defined analogously for a system with reliable FIFO links ($A(\mathrm{R})$), and a system with reliable non-FIFO links ($A(\mathrm{RN})$). The only difference in the case of a non-FIFO link is that the state of a link is expressed as a set and not a sequence.

5. *Results*

In this section, we present the results for the three link models using the notation we introduced. The formal proofs are provided later.

Consider a crashing protocol A for a graph $G = (V, E)$ consisting of a set V of nodes and a set E of links. The results for the three models are stated in terms of the subcomponents of the graph G which can be *controlled*—that is, which can be driven to some predetermined possible state in an execution. The remaining components are "beyond control"—that is, in that execution they attain some state that cannot be predetermined. In some sense, the power of the faults in the different fault models is characterized by the *maximal subset of the components of* G that can be controlled. We have informally described the fault spans of the three models in Figure 1. We now give a formal statement.

As described in the introduction, each node and each link of a protocol in the CAML model can be driven to any given possible state in an execution. In other words, the entire graph G can be controlled. The following result is called the Any State Theorem because it expresses the ability to drive a crashing protocol to any possible state.

Theorem 5.1 (Unreliable FIFO Any State). *Let A be an arbitrary crashing protocol, and let $[\mathbf{N}, \mathbf{L}]$ be any possible state of $A(U)$ for a graph $G = (V, E)$. Then*:

$$[\mathbf{N}_0, \mathbf{L}_0] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

In the CAM model, only the nodes can be controlled—that is, the subset of G consisting of only the nodes can be driven to a given possible state. We call the result for Reliable FIFO links the Any Node State Theorem because in this case each node can be driven to any possible state.

THEOREM 5.2 (RELIABLE FIFO ANY NODE STATE). *Let A be an arbitrary crashing protocol, and let* $\mathbf{N}$ *be any possible node state vector of $A(R)$ for a graph $G = (V, E)$. Then there exists a state* $[\mathbf{N}, \mathbf{L}]$ *of $A(R)$ such that*

$$[\mathbf{N}_0, \mathbf{L}_0] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

In the CAMO model, any acyclic subgraph of the components of G (including links as well as nodes) can be driven to a given possible state. The result for Reliable non-FIFO links is called the Any Acyclic State Theorem as in this case any acyclic subgraph of components can be driven to any possible state.

THEOREM 5.3 (RELIABLE NON-FIFO ANY ACYCLIC STATE). *Let A be an arbitrary crashing protocol, and consider $A(RN)$ for a graph $G = (V, E)$. Consider (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle wholly among elements of $V_a \cup E_a$. Consider also an assignment of any possible node state s^i for every node $i \in V_a$ and of any possible link state $s^{i,j}$ for every link $(i, j) \in E_a$. Then there exists a state* $[\mathbf{N}, \mathbf{L}]$ *of $A(RN)$ such that $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i, j] = s^{i,j}$, $(i, j) \in E_a$ and*

$$[\mathbf{N}_0, \mathbf{L}_0] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

6. *Applications*

In this section, we discuss the applications of the results described in Section 5. The results in Section 5 are of interest in their own right, describing the fault span of protocols under different fault models. However, we can also derive other interesting results regarding impossibility of certain protocols as a direct corollary of the results in Section 5. Some of these results are well known (the Data Link Impossibility result) and some are new (the Token Passing and Resource Allocation Impossibility results). However, all these results are easily derived by applying the same Unreliable FIFO Any State theorem, Theorem 5.1. Though these impossibility results are derived below for the most important case of a crashing system with Unreliable FIFO links, some of the results remain valid for the other link models as well. For example, the results regarding the Impossibility of Token Passing and Resource Allocation protocols are also valid in the weaker fault model of Reliable FIFO and Reliable non-FIFO links. We, however, present proofs only in the context of Unreliable FIFO links.

To prove the impossibility of a protocol, it suffices to prove that there cannot be a protocol which satisfies the required correctness criteria in the given model. The correctness of protocols can be specified either in terms of sets of *legal executions* or sets of *legal behaviors*. We describe how our theorem can be used to prove impossibility results for two examples: a token passing protocol (correctness in terms of executions) and a Data Link (correctness specified using behaviors). We also show how the token passing proof can be extended to showing that there is no crashing solution to the Dining or Drinking Philosophers problem.

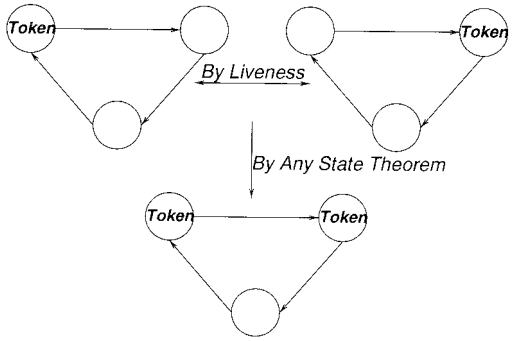

FIG. 15. Token passing impossibility.

6.1. TOKEN PASSING. We prove that it is impossible to have a reliable crashing token passing protocol. We first define a token passing protocol and state its correctness criteria. Then we prove impossibility using our first theorem.

We define a token passing protocol for a graph[6] $G = (V, E)$ as a crashing protocol (see Section 3) $T = (A^1, A^2, \ldots, A^n)$ where $n = |V|$, and where for all $i \in V$, there exists a function $token^i$ that maps the states of A^i to *true* or *false*. The function $token^i$ is used to indicate the presence (or absence) of a token in node A^i. Let $T(\mathrm{U})$ be a crashing automaton for T for graph $G = (V, E)$.

$T(\mathrm{U})$ is said to be correct if it satisfies two properties:

(**T1**) *Safety*. For all executions α of $T(\mathrm{U})$ and any state s which occurs in α, $token^i(s|A^i) = true$ for at most one $i \in V$.

(**T2**) *Liveness*. In any fair execution α of $T(\mathrm{U})$, for all $i \in V$, there exists infinitely many states s such that $token^i(s|A^i) = true$.

The first property says there are no duplicate tokens in any reachable state. The second property says that in any fair execution all nodes receive the token an infinite number of times. Note that the definition allows states in which no node has the "token"; for example, the "token" could be "on the links". The proof of the result given below is illustrated in Figure 15.

THEOREM 6.1 (TOKEN PASSING IMPOSSIBILITY). *There exists no correct crashing token passing protocol.*

PROOF. Let T be a correct crashing token passing protocol for graph G. Let $T(\mathrm{U})$ be the crashing automaton for T corresponding to a graph $G = (V, E)$. By liveness (**T2**), there is a fair execution α which contains states s_i and s_j such that $token^i(s_i|A^i) = true$ and $token^j(s_j|A^j) = true$ for $i \neq j$, $i, j \in V$. Consider the state s such that $s|A^i = s_i|A^i$ and $s|A^j = s_j|A^j$ and $s|A^k = s_0|A^k$ for $k \neq i, j$ and $k \in V$, and where s_0 is the initial state of $T(\mathrm{U})$. Clearly, s is a possible state of $T(\mathrm{U})$. Applying the Any State Theorem (Theorem 5.1), there exists a finite execution α' which takes $T(\mathrm{U})$ from state s_0 to state s, such that s violates the safety property **T1**. Thus, T is not correct. □

It is easy to adapt this proof to the other link models as well because we have only used the ability to control node states.

[6] There is no restriction on the topology to rings though the results are equally valid for token rings.

6.2. Resource Allocation Protocols. Mutual exclusion is closely related to the problem of resource allocation. Resource allocation problems (including k-exclusion, Dining Philosophers, or Drinking Philosophers [Lynch 1996]) can be described in terms of *exclusion sets*: an exclusion set is a collection of nodes that are not allowed to have simultaneous access to some critical resource (safety). For example, in the dining philosophers problem, processes that share a resource are connected by an edge in the topology graph. Thus, sets containing a node and its neighbors are exclusion sets. We model access to the resource by a Boolean function $critical^i(s)$ which is *true* if node i can access its critical section.

Assume that there is one such exclusion set $E \subseteq V$ with at least two nodes j and k. Assume there is a liveness condition which shows that for each node i, there is some execution which contains a state s_i such that $critical^i(s_i)$ is *true*. Then, exactly as in token passing, we can use the Any State Theorem to drive the resource allocation protocol to a state s in which both $critical^i(s)$ and $critical^j(s)$ are *true*. Thus, j and k are both in their respective critical sections in state s. This violates the safety property. We omit formal details.

6.3. Data Link Protocols. A *data link protocol* $D = (A^t, A^r)$ is a crashing protocol for a graph $G = (V, E)$ where $V = \{t, r\}$ and $E = \{(t, r), (r, t)\}$. Fix a message alphabet M. The node automaton A^t of D (transmitter automaton) has an additional input action $send\_msg(m)$, $m \in M$, and A^r (receiving automaton) has an additional output action $receive\_msg(m)$, $m \in M$. The action $send\_msg(m)$ is used by A^t to send a message m to A^r which receives the message by the action $receive\_msg(m)$. Let $D(\mathrm{U})$ be the crashing automaton for D and $G = (V, E)$. We only present an informal rendering of the proof. A formal proof can be found in Jayaram [1996].

Without crash actions, we would require that every message sent is received. The specification with crash actions is more delicate (see Fekete et al. [1993]), but it is sufficient to describe two reasonable correctness conditions that must be satisfied even with crashes. Refer to Fekete et al. [1993] for the complete formal correctness requirements. All behaviors of a correct data link automaton must, however, satisfy the following two conditions:

(**D1**) If a_i is a $send\_msg(m)$ action after which no crash action occurs in β, then there is a later $receive\_msg(m)$ action in β. (Intuitively, this is saying that after all crashes stop, all sent messages should be delivered.)

(**D2**) There is a correspondence function such that every $receive\_msg(m)$ action corresponds to exactly one earlier $send\_msg(m)$ action. (Intuitively, this is saying that any received message must correspond to a prior sent message.)

Define a *quiescent state* of the data link protocol D as a state after which there are no further *receive_msg* actions if there are no further input actions. Next, we claim that there is an execution α consisting of the sending and receiving of a single message m (and no other input actions) that ends with a quiescent state. Intuitively, if it did not end with a quiescent state, α could be extended and deliver more messages without any corresponding sends.

The impossibility proof is then illustrated in Figure 16. Let the quiescent state be s such that $s|A^t = a$, $s|A^r = b$, $s|U^{t,r} = Y$, $s|U^{r,t} = X$ as shown in the figure. Let i be the unique initial state of A^t. Let $G(a)$ be the sequence of packets received by the transmitter in α. Let s' be equal to s except that $s|A^t = i$ and

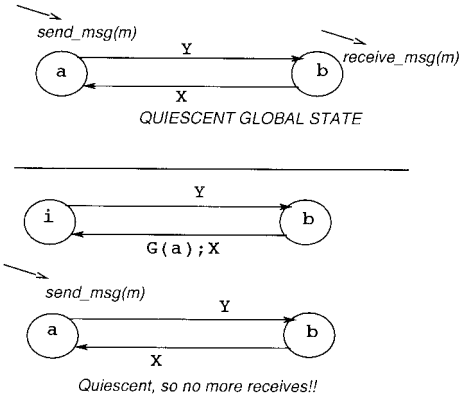

FIG. 16. Data link Impossibility.

$s'|U^{r,t} = G(a)\ X$ as shown in the figure. Because s' is a possible state, we can use the Any State Theorem to drive the protocol to state s'. Now we apply $acts^t(\alpha)$ to this state (this is the schedule corresponding to the transmitter actions in α). Clearly this schedule includes a *send_msg*(m) action with no further input actions, and results in the removal of $G(a)$ from the receiver-transmitter link. However, it can also result in the transmitter sending further packets; but we arrange for all these packets to be lost by the link. The result is the quiescent state s. We can then extend this execution to produce a fair execution in which there is a *send_msg*(m) with no subsequent *receive_msg*(m) or crash actions, a contradiction.

The Data Link impossibility result shows that any crashing Data Link protocol can be made to lose a message. However, our theorem can also be used to show the possibility of other, possibly more pernicious, failure modes. For example, most sliding window protocols use a sender window of say sequence numbers s to $s + w$, where s is the lower edge of the sender window, and w is the window size. Intuitively, this is the range of sequence numbers the sender is currently transmitting but has not received acknowledgements for. The receiver keeps track of the last number r it has received in sequence.

A basic invariant for correctness is that $r \in [s, s + w]$. However, our theorem shows that there is some sequence of crashes that can drive a standard Data Link protocol (which has no NVRAM) into a state such that $r \notin [s, s + w]$. For example, the receiver could be expecting sequence number 20 while the sender is sending sequence numbers in the range 1 through 17. This can lead to livelock with the sender's transmissions being persistently dropped at the receiver.

7. *Send and Receive Sequences*

In this section, we describe some additional notation that is used in the proofs in the next few sections. We believe that the vector notation combined with the additional terminology allows a natural and simple description of the construction underlying the proofs.

The following discussion applies to all three system models: a system with unreliable FIFO links $A(\mathrm{U})$, with reliable FIFO links $A(\mathrm{R})$, and with reliable non-FIFO links $A(\mathrm{RN})$.

The results are given in terms of states of nodes and sequences of packets sent or received in some execution of the system. We need further notation to describe such states and sequences.

If $\alpha = s_0\ a_1\ s_1\ a_2\ s_2 \cdots a_n\ s_n$ is a finite execution of the system, we will denote by α_k, $k \leq n$ the portion of the execution $s_0\ a_1\ s_1\ a_2\ s_2 \cdots a_k\ s_k$. Thus, $\alpha_n = \alpha$. Note that for all $k \leq n$, α_k is also an execution.

For any given finite execution $\alpha = s_0\ a_1\ s_1\ a_2\ s_2 \cdots a_n\ s_n$, let $\mathbf{NST}(\alpha)$ denote the node state vector corresponding to the final state s_n of the execution. Recall that $last\_state^j(\alpha)$ denotes $s_n|A^j$, the state of node A^j in the final state s_n.

Let the sequence of packets received on link (i, j) in any finite execution α be $rcv^{i,j}(\alpha)$. Also let $\mathbf{RCV}^{i,j}(\alpha)$ be the link state vector such that $\mathbf{RCV}^{i,j}(\alpha)[i, j] = rcv^{i,j}(\alpha)$ and $\mathbf{RCV}^{i,j}(\alpha)[k, l] = \epsilon$ for $(k, l) \neq (i, j)$. Let $\mathbf{RCV}^j(\alpha)$ be the link state vector such that $\forall i$ such that $(i, j) \in E$, $\mathbf{RCV}^j(\alpha)[i, j] = rcv^{i,j}(\alpha)$ and for every other (k, l), $k, l \neq j$, $\mathbf{RCV}^j(\alpha)[k, l] = \epsilon$. Informally, $\mathbf{RCV}^{i,j}$ is the vector (matrix) whose (i, j)th element contains $rcv^{i,j}(\alpha)$, and whose remaining elements are empty. Similarly, $\mathbf{RCV}^j$ is the matrix whose column j is filled in with $rcv^{1,j}(\alpha)$ through $rcv^{n,j}(\alpha)$, and whose remaining elements are empty. These and allied definitions allow us to reason completely in terms of vectors instead of sequences.

Recall from Section 4, that the concatenation of link state vectors for a FIFO link is defined as the concatenation of the corresponding sequences for each array element in the component link state vectors. Thus, $\mathbf{K} = \mathbf{L}\ \mathbf{M}$ is the *concatenation of link state vectors* $\mathbf{L}$ and $\mathbf{M}$ if $\mathbf{K}[i, j] =$ the sequence $\mathbf{L}[i, j]\mathbf{M}[i, j]$, for all $(i, j) \in E$. To allow convenient concatenation of a larger set of link state vectors, we also introduce the concatenation operator Π, where $\Pi_{\forall i \in S}\mathbf{L_i}$ denotes the concatenation of all the link state vectors $\mathbf{L_i}$ defined by some index set S to which i belongs, and where the link state vectors are concatenated in lexicographic order based on the index. S will typically represent V the set of nodes, or E the set of edges.

Using this operator, note that $\mathbf{RCV}^j(\alpha = \Pi_{\forall (i,j) \in E}\mathbf{RCV}^{i,j}(\alpha)$. Intuitively, $\mathbf{RCV}^j(\alpha)$ is a vector containing all the packets received by j on *all* its incident links in execution α, with the other links being empty.

Finally for receive sequences let $\mathbf{RCV}(\alpha)$ be the link state vector such that $\forall (i, j) \in E$, $\mathbf{RCV}(\alpha)[i, j] = rcv^{i,j}(\alpha)$. Note that $\mathbf{RCV}(\alpha) = \Pi_{\forall j \in V}\mathbf{RCV}^j(\alpha)$. Intuitively, $\mathbf{RCV}(\alpha)$ is a vector containing all the packets received by all nodes on *all* incident links in execution α.

We define the terms for send sequences analogous to the terms for receive sequences. Let the sequence of packets sent on link (j, k) in any finite execution α be $snd^{j,k}(\alpha)$. For $(j, k) \in E$, let $\mathbf{SND}^{j,k}(\alpha)$ be the link state vector such that $\mathbf{SND}^{j,k}(\alpha)[j, k] = snd^{j,k}(\alpha)$ and $\mathbf{SND}^{j,k}(\alpha[l, m] = \epsilon$ for $(l, m) \neq (j, k)$. Let $\mathbf{SND}^j(\alpha)$ be the link state vector such that $\forall k$ such that $(j, k) \in E$, $\mathbf{SND}^j(\alpha)[j, k] = snd^{j,k}(\alpha)$ and for all other (l, m), $\mathbf{SND}^j(\alpha)[l, m] = \epsilon$. Note that $\mathbf{SND}^j(\alpha) = \Pi_{\forall (j,k) \in E}\mathbf{SND}^{j,k}(\alpha)$. Finally let $\mathbf{SND}(\alpha)$ be the link state vector such that $\forall (j, k) \in E$, $\mathbf{SND}(\alpha)[j, k] = snd^{j,k}(\alpha)$. Note that $\mathbf{SND}(\alpha) = \Pi_{\forall j \in V}\mathbf{SND}^j(\alpha)$. Note also the difference between say $\mathbf{SND}^j(\alpha)$ and $\mathbf{RCV}^j(\alpha)$; the former denotes what is sent from j to all its neighbors, while the latter denotes what is received by j from all neighbors.

Figure 17 illustrates the difference between send and receive sequences. We note the following relationship between $rcv^{i,j}(\alpha)$ and $snd^{i,j}(\alpha)$. If link (i, j) is

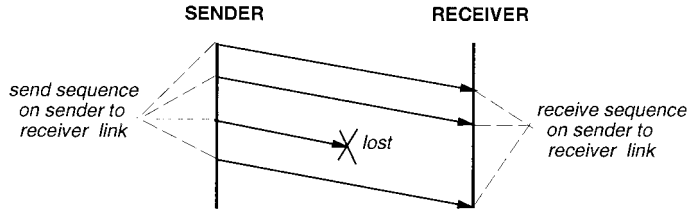

FIG. 17. Send and receive sequences. Note that the receive sequence does not include any packets that were sent and lost on the link.

unreliable FIFO, then $rcv^{i,j}(\alpha)$ is a subsequence of $snd^{i,j}(\alpha)$. This follows because only a packet that has been sent in a link in an execution can be received from that link in that execution. Moreover, as the link is FIFO and lossy, the received packets may be any subsequence of the packets which were sent. If the link (i, j) is reliable FIFO, then $rcv^{i,j}(\alpha)$ is strictly a prefix of $snd^{i,j}(\alpha)$. First, packets are received in order because the link is FIFO. Second, because the link is not lossy, the packets received must be a prefix of the packets sent. Finally, if the link (i, j) is reliable non-FIFO, then the set of packets in $rcv^{i,j}(\alpha)$ is a subset of the set of packets in $snd^{i,j}(\alpha)$. The reasoning is the same as for reliable FIFO links. However, because the link is non-FIFO, packets may not be received in the order in which they were sent.

8. *Proofs for Unreliable FIFO*

In this section, we present the proofs of the result for unreliable FIFO links. Our proof for general graphs uses a different inductive technique (than the intuitive proof described for two node graphs earlier) to build up a sequence on a link. Rather than build up a sequence on a single link, it builds a sequence on *all* links at the same time. This will become clear as the construction is outlined.

Our first three lemmas capture the three main facets of the CAML model. The first lemma captures the effect of allowing Crashes without nonvolatile memory; the second lemma captures the effect of allowing arbitrary packet loss on a link; the third lemma captures the effect of asynchrony. We start with a Crash lemma which is true for all three link models. The Crash lemma essentially states that from any global state, we can always change the state of node i to its initial state by crashing node i.

LEMMA 8.1 (CRASH). *Consider any crashing automaton $A(U)$, $A(R)$, or $A(RN)$ for graph G. Then for all states* $[\mathbf{N}, \mathbf{L}]$, $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}', \mathbf{L}]$ *where* $\mathbf{N}'$ *is the same as* $\mathbf{N}$ *except that for some* $i \in V$, $\mathbf{N}'[i]$ = *the unique start state of* A^i.

PROOF. Let X represent the crashing automaton. We must prove that if there exists a finite execution α of X such that the final state of α is $[\mathbf{N}, \mathbf{L}]$, then there exists a finite extension α' of α such that the final state of α' is $[\mathbf{N}', \mathbf{L}]$. Let the last state of α be s_n. Then we simply extend α to α' by adding the action $crash^i$ and the resulting state, say s_{n+1}. This is a valid extension because X is input enabled, and so $crash^i$ is enabled at s_n. It is easy to see that s_{n+1} is the same as s_n except that $s_{n+1}|A^i$ is the unique start state of A^i. □

We now state the loss lemma for unreliable links that shows that for any global state and any link (i, j), we can lose an arbitrary subsequence of packets on the link.

LEMMA 8.2 (UNRELIABLE FIFO LOSS). *Consider a protocol $A(U)$ for a graph $G = (V, E)$. Let* **L** *be a link state vector such that for some $(i, j) \in E$, $\mathbf{L}[i, j] = Q$, where Q is a finite sequence of packets and Q' is a subsequence of Q. Then for all node vectors* **N**, $[\mathbf{N}, \mathbf{L}] \overset{\beta}{\rightsquigarrow} [\mathbf{N}, \mathbf{L}']$, *where* $\mathbf{L}' = \mathbf{L}$ *except that* $\mathbf{L}'[i, j] = Q'$ *and* β *consists only of* $lose^{i,j}$ *actions.*

PROOF. The proof essentially consists of exhibiting a schedule consisting of $lose^{i,j}$ actions corresponding to the packets in Q that are not in Q'. The only delicate point is that we cannot lose a packet if its attempt counter is in the *keep* set for that packet (see model for a live unreliable link in Section 3). But that can easily be handled by considering an equivalent execution that results in the same state $[\mathbf{N}, \mathbf{L}]$ but which begins with values in the *keep* array that allow us to lose the desired packets. We omit formal details. □

We now state and prove a reliable FIFO Rotate Node lemma. This lemma states that if a node j is supplied with *all* the packets it receives in some execution α, then node j can go to its corresponding state at the end of α, and can be made to send all the packets it sends in α. All other packets in the links (formalized by a link state vector **O**) remain in the links, except that the **O** packets "rotate" to the head of each link, and the sequences sent by j rotate to the tail of the links outgoing from node j. This lemma expresses the effect of asynchrony and locality in our model.

LEMMA 8.3 (UNRELIABLE FIFO ROTATE NODE). *Consider a finite execution α of $A(U)$. For all node state vectors* **N** *and link state vectors* **O** *of $A(U)$ and for any $j \in V$,*

$$[\mathbf{N}, \mathbf{RCV}^j(\alpha)\mathbf{O}] \overset{\beta}{\rightsquigarrow} [\mathbf{N}', \mathbf{O}\ \mathbf{SND}^j(\alpha)]$$

where $\beta = crash^j\ acts^j(\alpha)$ and $\mathbf{N}'$ *is the same as* **N** *except that* $\mathbf{N}'[j] = last\_state^j(\alpha)$.

PROOF. The proof essentially works by first crashing j and then having j receive all the packets stored in $\mathbf{RCV}^j(\alpha)$. This causes j to emit all the packets in $\mathbf{SND}^j(\alpha)$ and end up in the state corresponding to the last state of execution α. The reader who finds this intuitively clear may wish to skip the following paragraph on a first reading.

We will verify that each node and link will be in the desired final state after applying β. Consider node j first and $\beta|A^j = crash^j\ acts^j(\alpha)$. Now $crash^j$ takes A^j to its start state as A^j is crashing and $\beta|A^j$ is the sequence of actions in an execution fragment which starts at the initial state and takes A^j to $last\_state^j(\alpha) = \mathbf{N}'[j]$. Next, for all other nodes $i \neq j$, $\beta|A^i$ is the empty sequence which leaves A^i at $\mathbf{N}[i] = \mathbf{N}'[i]$.

Next, consider the links. First, consider the links for which node j is the receiver. Thus, for any i such that $(i, j) \in E$, $\beta|\mathrm{U}^{i,j} = receive^{i,j}(p_1)receive^{i,j}(p_2) \cdots receive^{i,j}(p_l)$ where $rcv^{i,j}(\alpha) = p_1 p_2 \cdots p_l$. If $\mathbf{L}' = \mathbf{O}\ \mathbf{SND}^j(\alpha)$, then $\beta|\mathrm{U}^{i,j}$ takes $\mathrm{U}^{i,j}$ to $\mathbf{O}[i, j] = \mathbf{L}'[i, j]$. Next consider any links for which node j is the

sender. Thus, for any k such that $(j, k) \in E$, $\beta|U^{j,k} = send^{j,k}(p'_2)send^{j,k}(p'_2) \cdots send^{j,k}(p'_m)$, where $snd^{j,k}(\alpha) = p'_1p'_2 \cdots p'_m$. This takes $U^{j,k}$ to $\mathbf{L}'[j, k]$. Next, consider all other links. For $(u, v) \in E$, $u, v \neq j$, $\beta|U^{u,v}$ is the empty sequence which thus leaves $U^{u,v}$ at $\mathbf{O}[u, v] = \mathbf{L}'[u, v]$. Note that $\mathbf{RCV}^j(\alpha)[u, v] = \epsilon$.

Thus, we have verified that β results in all nodes and links being in the states specified by $[\mathbf{N}', \mathbf{O}\ \mathbf{SND}^j(\alpha)]$. □

The next corollary states that by loading the links with the receive sequences for all nodes in some execution α, we can cause every node j to emit the sequence of packets that j emits in α. Once again, the old packets remain unchanged.

COROLLARY 8.4 (UNRELIABLE FIFO ROTATE). *Consider a finite execution α of $A(U)$. For all node state vectors* **N** *and link state vectors* **O** *of $A(U)$,*

$$[\mathbf{N}, \mathbf{RCV}(\alpha)\mathbf{O}] \rightsquigarrow [\mathbf{N}', \mathbf{O}\ \mathbf{SND}(\alpha)],$$

where $\forall j \in V$, $\mathbf{N}'[j] = last\_state^j(\alpha)$.

PROOF. This corollary is proved by rewriting $\mathbf{RCV}(\alpha)$ as $\Pi_{\forall j \in V}\mathbf{RCV}^j(\alpha)$, applying the Rotate Node lemma (Lemma 8.3) in turn to each node $j \in V$. This results in the link vector $\Pi_{\forall j \in V}\mathbf{SND}^j(\alpha)$ which can be rewritten as $\mathbf{SND}(\alpha)$. □

The next lemma states that we can obtain the receive sequences corresponding to an execution from the corresponding send sequences by losing the appropriate packets. Recall that α_k refers to the prefix of execution α that ends with the $k + 1$-st state in α. (If $\alpha = s_0\ a_1\ s_1\ a_2\ s_2 \cdots a_n\ s_n$, then α_k ends with state s_k.)

LEMMA 8.5 (UNRELIABLE FIFO SEND TO RECEIVE). *Consider an execution α of $A(U)$ of length n. For all node state vectors* **N** *and link state vectors* **O** *and for $0 \leq k < n$,*

(a) $[\mathbf{N}, \mathbf{SND}(\alpha_k)\ \mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{RCV}(\alpha_{k+1})\ \mathbf{O}]$.
(b) $[\mathbf{N}, \mathbf{SND}(\alpha_k)\ \mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{RCV}(\alpha_k)\ \mathbf{O}]$.

PROOF. For every link $U^{i,j}$, $(i, j) \in E$, $rcv^{i,j}(\alpha_{k+1})$ is a subsequence of $snd^{i,j}(\alpha_k)$. This follows because the link $U^{i,j}$ is unreliable FIFO and all packets which are received on a link in an execution must be sent on the link before the last receive. Thus by applying the Unreliable FIFO Loss lemma (Lemma 8.2) on links $U^{i,j}$, $\forall (i, j) \in E$, we prove part (a) that $[\mathbf{N}, \mathbf{SND}(\alpha_k)\mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{RCV}(\alpha_{k+1})\mathbf{O}]$. For every link $U^{i,j}$, $(i, j) \in E$, $rcv^{i,j}(\alpha_k)$ is a subsequence of $rcv^{i,j}(\alpha_{k+1})$ and hence a subsequence of $snd^{i,j}(\alpha_k)$. Thus, the proof for part (b) is similar to the proof for part (a). □

The next lemma shows that we can "pump" up the send sequence (on all links) corresponding to a prefix α_k of execution α, and produce a send sequence corresponding to a longer prefix α_{k+1}. Once again, old packets "rotate" to the front of the links.

LEMMA 8.6 (UNRELIABLE FIFO ROTATE WITH INCREMENT). *Consider an execution α of $A(U)$ of length n. For all node state vectors* **N** *and link state vectors* **O**

and for $k < n$,

$$[\mathbf{N}, \mathbf{SND}(\alpha_k)\mathbf{O}] \rightsquigarrow [\mathbf{N}', \mathbf{O}\ \mathbf{SND}(\alpha_{k+1})];$$

where for all $j \in V$, $\mathbf{N}'[j] = last\_state^j(\alpha_{k+1})$.

PROOF. By the Send to Receive lemma (Lemma 8.5(a)), we get,

$$[\mathbf{N}, \mathbf{SND}(\alpha_k)\mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{RCV}(\alpha_{k+1})\mathbf{O}].$$

By the Rotate corollary (Corollary 8.4), we get,

$$[\mathbf{N}, \mathbf{RCV}(\alpha_{k+1})\mathbf{O}] \rightsquigarrow [\mathbf{N}', \mathbf{O}\ \mathbf{SND}(\alpha_{k+1})].$$

Thus, the lemma is proved by transitivity of transitions. □

In what follows, recall that $\mathbf{N_0}$ is the node state vector where for all $i \in V$, $\mathbf{N_0}[i]$ = unique start state of node A^i.

LEMMA 8.7 (UNRELIABLE FIFO ROTATE WITHOUT INCREMENT). *Consider an execution* α *of* $A(U)$ *of length* n. *For all node state vectors* **N** *and link state vectors* **O** *and for* $k < n$,

$$[\mathbf{N}, \mathbf{SND}(\alpha_k)\mathbf{O}] \rightsquigarrow [\mathbf{N_0}, \mathbf{O}\ \mathbf{SND}(\alpha_k)].$$

PROOF. By the Send to Receive lemma (Lemma 8.5(b)), we get

$$[\mathbf{N}, \mathbf{SND}(\alpha_k)\mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{RCV}(\alpha_k)\mathbf{O}].$$

By the Rotate corollary (Corollary 8.4), we get,

$$[\mathbf{N}, \mathbf{RCV}(\alpha_k)\mathbf{O}] \rightsquigarrow [\mathbf{N}', \mathbf{O}\ \mathbf{SND}(\alpha_k)].$$

Also, by Lemma 8.1, applied to all $i \in V$,

$$[\mathbf{N}', \mathbf{O}\ \mathbf{SND}(\alpha_k)] \rightsquigarrow [\mathbf{N_0}, \mathbf{O}\ \mathbf{SND}(\alpha_k)].$$

Thus, the lemma is proved by transitivity of transitions. □

For the next lemma, we define a link state vector **L** to be a *concatenation of send sequences* for $A(\mathrm{U})$, if there exist finite executions $\alpha^1, \alpha^2, \ldots, \alpha^m$ of $A(\mathrm{U})$ such that: $\mathbf{L} = \mathbf{SND}(\alpha^1)\mathbf{SND}(\alpha^2) \cdots \mathbf{SND}(\alpha^m)$.

Our next lemma states that any send sequence that is sandwiched in between two link state vectors **O** and **O′** can be rotated to the leftmost position as long as both **O** and **O′** are concatenations of send sequences.

LEMMA 8.8 (UNRELIABLE FIFO ANY ROTATE). *Let* **O** *and* **O′** *each be a concatenation of send sequences for* $A(U)$. *Let* α *be a finite execution of* $A(U)$. *Let* $\mathbf{L} = \mathbf{O}\ \mathbf{SND}(\alpha)\mathbf{O}'$ *and* $\mathbf{L}' = \mathbf{SND}(\alpha)\ \mathbf{O}'\ \mathbf{O}$. *Then*

$$[\mathbf{N_0}, \mathbf{L}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}'].$$

PROOF. We start by rewriting **O** and **O′** as a concatenation of send sequences (possibly from different executions.) We are then left with a link state vector that is a concatenation of send sequences which contains $\mathbf{SND}(\alpha)$. To rotate a concatenation of send sequences one step to the left (so that the leftmost send

sequence moves to the right), we simply apply the Unreliable FIFO Rotate without Increment lemma (Lemma 8.7) to the leftmost send sequence. We keep rotating one step at a time until $\mathbf{SND}(\alpha)$ is in the leftmost position.

More formally, by the definition of a send sequence, we can rewrite $\mathbf{O} = \mathbf{SND}(\alpha^1) \cdots \mathbf{SND}(\alpha^k)$ where $\alpha^1 \cdots \alpha^k$ are different executions. Similarly, we can rewrite $\mathbf{O'} = \mathbf{SND}(\alpha^{k+1}) \cdots \mathbf{SND}(\alpha^n)$, where $\alpha^{k+1} \cdots \alpha^n$ are also different executions.

Thus

$$\mathbf{L} = \mathbf{SND}(\alpha^1) \cdots \mathbf{SND}(\alpha^k)\mathbf{SND}(\alpha)\mathbf{SND}(\alpha^{k+1}) \cdots \mathbf{SND}(\alpha^n).$$

We are going to rotate the first k send sequences to the left. To this end, define the result of doing the first $i - 1$ rotations for $1 \leq i \leq k$ as

$$\mathbf{L_i} = \mathbf{SND}(\alpha^i) \cdots \mathbf{SND}(\alpha^k)\mathbf{SND}(\alpha)\mathbf{SND}(\alpha^{k+1}) \cdots \mathbf{SND}(\alpha^n)\mathbf{SND}(\alpha^1) \cdots \mathbf{SND}(\alpha^{i-1}),$$

and define

$$\mathbf{L_{k+1}} = \mathbf{L'} = \mathbf{SND}(\alpha)\mathbf{SND}(\alpha^{k+1}) \cdots \mathbf{SND}(\alpha^n)\mathbf{SND}(\alpha^1) \cdots \mathbf{SND}(\alpha^k).$$

Using Lemma 8.7, it follows that for $i \leq k$,

$$[\mathbf{N_0}, \mathbf{L_i}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{i+1}}].$$

Applying the last relation repeatedly for $i = 1, \cdots k$ and using transitivity, we have

$$[\mathbf{N_0}, \mathbf{L_1}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{k+1}}].$$

The lemma follows because $\mathbf{L_1} = \mathbf{L}$ and $\mathbf{L_{k+1}} = \mathbf{L'}$. □

Consider a send sequence $\mathbf{SND}(\alpha^k)$ corresponding to a prefix of an execution α that is sandwiched in between two link state vectors $\mathbf{O}$ and $\mathbf{O'}$. Suppose both $\mathbf{O}$ and $\mathbf{O'}$ are a concatenation of send sequences. Then $\mathbf{SND}(\alpha^k)$ can be pumped up to $\mathbf{SND}(\alpha^{k+1})$ without changing $\mathbf{O}$ and $\mathbf{O'}$, or their relative positions.

LEMMA 8.9 (UNRELIABLE FIFO INCREMENT). *Let* $\mathbf{O}$ *and* $\mathbf{O'}$ *each be a concatenation of send sequences for* $A(U)$. *Let* α *be a finite execution of* $A(U)$ *of length* n. *Let* $\mathbf{L} = \mathbf{O}\ \mathbf{SND}(\alpha_k)\ \mathbf{O'}$ *for* $0 \leq k < n$. *Let* $\mathbf{L'} = \mathbf{O}\ \mathbf{SND}(\alpha_{k+1})\ \mathbf{O'}$. *Then*

$$[\mathbf{N_0}, \mathbf{L}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L'}].$$

PROOF. The proof proceeds in three steps. We first rotate $\mathbf{L}$, to bring $\mathbf{SND}(\alpha_k)$ to the head. Then we rotate and increment $\mathbf{SND}(\alpha_k)$ to $\mathbf{SND}(\alpha_{k+1})$ and crash each node to bring the nodes to the initial state. Finally, we rotate back $\mathbf{SND}(\alpha_{k+1})$ to the original position.

More formally, by the Unreliable FIFO Any Rotate lemma (Lemma 8.8)

$$[\mathbf{N_0}, \mathbf{L}] \rightsquigarrow [\mathbf{N_0}, \mathbf{SND}(\alpha_k)\mathbf{O'O}].$$

By the Unreliable FIFO Rotate with Increment lemma (Lemma 8.6)

$$[\mathbf{N_0}, \mathbf{SND}(\alpha_k)\mathbf{O'O}] \rightsquigarrow [\mathbf{N_0}, \mathbf{SND}(\alpha_{k+1})\mathbf{O'O}].$$

Finally, by the Unreliable FIFO Any Rotate lemma (Lemma 8.8)

$$[\mathbf{N_0}, \mathbf{SND}(\alpha_{k+1})\mathbf{O'O}] \rightsquigarrow [\mathbf{N_0}, \mathbf{O\ SND}(\alpha_{k+1})\mathbf{O'}].$$

The lemma follows from the last three relations and transitivity. □

LEMMA 8.10 (UNRELIABLE FIFO CONCATENATION). *Let* **L** *be any concatenation of send sequences for* $A(U)$. *Then*

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}].$$

PROOF. From the definition, we can always rewrite **L** as:

$$\mathbf{L} = \mathbf{SND}(\alpha^1)\mathbf{SND}(\alpha^2) \cdots \mathbf{SND}(\alpha^m),$$

where $\alpha^1, \alpha^2, \ldots, \alpha^m$ are finite executions of $A(\mathrm{U})$. Let n_i be the index of the last state in α^i.

For any execution α and for any link $\mathrm{U}^{i,j}$, $(i, j) \in E$, $rcv^{i,j}(\alpha_1) = \epsilon$. This follows because the first action of an execution cannot be a *receive*. Thus

$$\mathbf{L_0} = \mathbf{RCV}(\alpha_1^1)\mathbf{RCV}(\alpha_1^2) \cdots \mathbf{RCV}(\alpha_1^m).$$

So consider

$$\mathbf{L'} = \mathbf{SND}(\alpha_1^1)\mathbf{SND}(\alpha_1^2) \cdots \mathbf{SND}(\alpha_1^m).$$

We can prove that $[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L'}]$. This is proved by applying the Unreliable FIFO Rotate Corollary (Corollary 8.4) to transform each **RCV** into a corresponding **SND**.

More formally, define for $1 \leq i \leq m$,

$$\mathbf{L_i} = \mathbf{RCV}(\alpha_1^{i+1}) \cdots \mathbf{RCV}(\alpha_1^m)\mathbf{SND}(\alpha_1^1) \cdots \mathbf{SND}(\alpha_1^i).$$

It follows from the Unreliable FIFO Rotate Corollary (Corollary 8.4) that

$$[\mathbf{N_0}, \mathbf{L}_i] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{i+1}}].$$

By applying this last relation repeatedly for $i = 1, \ldots m - 1$ and using transitivity, we get

$$[\mathbf{N_0}, \mathbf{L_1}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_m}].$$

But $\mathbf{L_1} = \mathbf{L_0}$ and $\mathbf{L_m} = \mathbf{L'}$.

From the last two relations and transitivity, we get

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L'}], \tag{1}$$

where

$$\mathbf{L'} = \mathbf{SND}(\alpha_1^1)\mathbf{SND}(\alpha_1^2) \cdots \mathbf{SND}(\alpha_1^m).$$

So far, we have shown how to transform the null sequence on each link to a concatenation of send sequences, where each send sequence is the length 1 prefix of the corresponding send sequence in **L**. Next, to prove the lemma we iteratively build up each of the send sequences in **L′** to its full length in **L**. We do so by using the Unreliable FIFO Increment Lemma (Lemma 8.9) n_i times on the **SND** that corresponds to α^i.

To this end, define $\mathbf{L_{k,j}}$, for $1 \leq k \leq m$, $1 \leq j \leq n_k$, as

$$\mathbf{L_{k,j}} = \mathbf{SND}(\alpha^1) \cdots \mathbf{SND}(\alpha^{k-1})\mathbf{SND}(\alpha_j^k)\mathbf{SND}(\alpha_1^{k+1}) \cdots \mathbf{SND}(\alpha_1^m).$$

For convenience, also define $\mathbf{L_{m+1,1}} = \mathbf{L}$. By the Unreliable FIFO Increment Lemma (Lemma 8.9), it follows that for $j = 1, \ldots n_k - 1$, $[\mathbf{N_0}, \mathbf{L_{k,j}}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{k,j+1}}]$. Thus, applying this relation repeatedly and by transitivity we get $[\mathbf{N_0}, \mathbf{L_{k,1}}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{k,n_k}}]$. But it follows from the definitions that $\mathbf{L_{k,n_k}} = \mathbf{L_{k+1,1}}$. Thus, from the last two relations and transitivity, it follows that for $0 \leq k \leq m - 1$

$$[\mathbf{N_0}, \mathbf{L_{k,1}}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{k+1,1}}].$$

Applying this last relation repeatedly for $k = 1 \ldots m$ and using transitivity yields

$$[\mathbf{N_0}, \mathbf{L_{1,1}}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L_{m+1,1}}].$$

But since $\mathbf{L_{1,1}} = \mathbf{L'}$ and $\mathbf{L_{m+1,1}} = \mathbf{L}$ the last relation can be rewritten as

$$[\mathbf{N_0}, \mathbf{L'}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}].$$

The lemma follows by using the last relation together with Eq. 1 and transitivity. □

For the main theorem, we need to define the *generator* corresponding to a node state vector and the *generator* corresponding to a link state vector. Informally, a generator is a link state vector that can drive the state of the network to a specified state.

Consider any node state vector **N** that is a possible node state vector. Then we say that **G** is a *node state generator* for **N** if there exists executions α^i of $A(\mathrm{U})$ such that $\forall i \in V$, $\mathbf{N}[i] = \mathit{last\_state}^i(\alpha^i)$ and $\mathbf{G} = \Pi_{\forall i \in V}\mathbf{SND}(\alpha^i)$. Intuitively, **G** has within it sequences for driving each node i to the state $\mathbf{N}[i]$. This can be formalized by the following lemma:

LEMMA 8.11 (UNRELIABLE FIFO NODE STATE GENERATION). *There exists some node state generator* **G** *for any possible node state vector* **N** *such that* **G** *is a concatenation of send sequences. Also, for any link state vector* **O** *and any node state vector* **N′**

$$[\mathbf{N'}, \mathbf{G}\ \mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{O}].$$

PROOF. If **N** is a possible node state vector, then for all $i \in V$, $\mathbf{N}[i] = \mathit{last\_state}^i(\alpha^i)$, for some executions α^i of $A(\mathrm{U})$. Thus, we set $\mathbf{G} = \Pi_{\forall i \in V}\mathbf{SND}(\alpha^i)$. This proves the first part of the lemma.

To prove the second part, we use the fact that $\mathbf{G} = \Pi_{\forall i \in V}\mathbf{SND}(\alpha^i)$ for some set of executions α^i such that $\forall i \in V$, $\mathbf{N}[i] = \mathit{last\_state}^i(\alpha^i)$. Consider $\mathbf{G'} =$

$\Pi_{\forall i \in V}\mathbf{RCV}^i(\alpha^i)$. Because for all $(i, j) \in E$, $\mathbf{G}'[i, j]$ is a subsequence of $\mathbf{G}[i, j]$, we can use the Unreliable FIFO Loss lemma (Lemma 8.2) to get

$$[\mathbf{N}', \mathbf{G}\ \mathbf{O}] \rightsquigarrow [\mathbf{N}', \mathbf{G}'\ \mathbf{O}]. \tag{2}$$

We now intuitively apply the following 2-step procedure to the receive sequence $\mathbf{RCV}^i(\alpha^i)$ that is currently at the leftmost position in the link state vector. This will result in removing $\mathbf{RCV}^i(\alpha^i)$ and driving the state of node i to $\mathbf{N}[i]$. If we iterate this procedure for all $i \in V$, we achieve the desired goal.

So let $\mathbf{RCV}^i(\alpha^i)$ be currently at the leftmost position in the link state vector. Intuitively, the two-step procedure is as follows:

—We apply the Unreliable FIFO Rotate Node lemma (Lemma 8.3) to drive the state of node i to $\mathbf{N}[i]$. The above is possible because $rcv^{j,i}(\alpha^i)$ is at the head of each link $(j, i) \in E$.

—The previous step drives node i to the goal state and removes the $\mathbf{RCV}(\alpha^i)$ sequence as desired. However, it also adds $\mathbf{SND}^i(\alpha^i)$ to the right end of the link state vector. However, this extraneous component can easily be removed by the Unreliable FIFO Link Loss Lemma (Lemma 8.2).

Thus, by iterating this procedure for all $i \in V$, we drive all node states to the corresponding state in $\mathbf{N}$ and we remove all the components in $\mathbf{G}$. We also leave the link state vector $\mathbf{O}$ unchanged: thus, we achieve the desired goal by the transitivity of transitions.

To capture this procedure formally, define $\mathbf{N_k}$, $0 \leq k \leq n$, such that $\mathbf{N_k}[i] = \mathbf{N}[k]$ for $i \leq k$ and $\mathbf{N_k}[i] = \mathbf{N}'[k]$ for $i > k$. (In other words, $\mathbf{N_k}$ is a node state vector in which the first k nodes are in the desired goal state specified by $\mathbf{N}$ and the remaining nodes are in their initial state specified by $\mathbf{N}'$.) Observe that $\mathbf{N_0} = \mathbf{N}'$ and $\mathbf{N_n} = \mathbf{N}$.

We also define for $0 \leq k \leq n$, $\mathbf{G_k} = \Pi_{\forall i > k}\mathbf{RCV}^i(\alpha^i)\ \mathbf{O}$. (In other words, this is the same as $\mathbf{G}'\ \mathbf{O}$ except that the first k **RCV** sequences have been removed.) Observe that $\mathbf{G_0} = \mathbf{G}'\ \mathbf{O}$ and $\mathbf{G_n} = \mathbf{O}$.

Using these two definitions, Lemma 8.3 and Lemma 8.2, we have for $0 \leq k \leq n - 1$, $[\mathbf{N_k}, \mathbf{G_k}] \rightsquigarrow [\mathbf{N_{k+1}}, \mathbf{G_{k+1}}]$. Thus, applying this relation repeatedly, we get $[\mathbf{N_0}, \mathbf{G_0}] \rightsquigarrow [\mathbf{N_n}, \mathbf{G_n}]$. Using the observations given above, this can be rewritten as $[\mathbf{N}', \mathbf{G}'\ \mathbf{O}] \rightsquigarrow [\mathbf{N}, \mathbf{O}]$.

The lemma follows by using the last relation together with Eq. 2 and transitivity. □

We now make a similar definition and lemma for link state generators. We say that $\mathbf{G}$ is a *link state generator* for a link state vector $\mathbf{L}$ if for all $(i, j) \in E$, $\mathbf{L}[i, j]$ is a subsequence of $\mathbf{G}[i, j]$. Essentially, a link state generator has all the packets required to produce a given link state vector (by losing the appropriate packets).

LEMMA 8.12 (UNRELIABLE FIFO LINK STATE GENERATION). *There exists some link state generator* $\mathbf{G}$ *for any possible link state vector* $\mathbf{L}$ *such that* $\mathbf{G}$ *is a concatenation of send sequences. Also, for any node state vector* $\mathbf{N}$,

$$[\mathbf{N}, \mathbf{G}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

PROOF. As $\mathbf{L}$ is a possible link state vector then for all $(i, j) \in E$, $\mathbf{L}[i, j]$ is a sequence of possible packets $= p_1 p_2 \cdots p_n$ such that $\forall k$, $p_k \in snd^{i,j}(\alpha_k^{i,j})$ for

some execution $\alpha_k^{i,j}$ of $A(\mathrm{U})$. Let $\mathbf{G_{i,j}} = \Pi_{\forall k}\mathbf{SND}(\alpha_k^{i,j})$. Then let us call $\mathbf{G} = \Pi_{\forall (i,j)\in E}\mathbf{G_{i,j}}$ the link state generator for $\mathbf{L}$. Note that $\mathbf{L}[i, j]$ is a subsequence of $\mathbf{G_{i,j}}[i, j]$, and thus $\mathbf{L}[i, j]$ is a subsequence of $\mathbf{G}[i, j]$. Thus, $\mathbf{G}$ is a link state generator for $\mathbf{L}$.

The second part of the lemma now follows directly from the Unreliable FIFO Link Loss Lemma (Lemma 8.2) because the goal sequence on each link (i, j), $\mathbf{L}[i, j]$, is a subsequence of $\mathbf{G}[i, j]$. □

We now show that if we can load the links with a node state generator for $\mathbf{N}$ and a link state generator for $\mathbf{L}$, then we can drive the system to $[\mathbf{N}, \mathbf{L}]$.

LEMMA 8.13 (UNRELIABLE FIFO PLAYOUT). *Let A be an arbitrary crashing protocol, and let $[\mathbf{N}, \mathbf{L}]$ be any possible state of $A(U)$ for a graph $G = (V, E)$. Suppose $\mathbf{G_N}$ and $\mathbf{G_L}$ are a node state generator for $\mathbf{N}$ and a link state generator for $\mathbf{L}$ respectively. Suppose further that $\mathbf{G_N}$, $\mathbf{G_L}$ are both a concatenation of send sequences. Then*:

$$[\mathbf{N_0}, \mathbf{G_N G_L}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

PROOF. Using the Unreliable FIFO Node State Generation Lemma, Lemma 8.11, we see that

$$[\mathbf{N_0}, \mathbf{G_N G_L}] \rightsquigarrow [\mathbf{N}, \mathbf{G_L}].$$

Using the Unreliable FIFO Link State Generation Lemma, Lemma 8.12, we see that

$$[\mathbf{N}, \mathbf{G_L}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

The proof follows from the transitivity of transitions. □

We are now ready to prove our main theorem for unreliable links, which states that we can drive the system to any possible global state. The proof uses the earlier concatenation construction to load the links with a node and link state generator for the goal state; it then uses the previous lemma to play out these generators to result in the goal state.

THEOREM 8.14 (UNRELIABLE FIFO ANY STATE). *Let A be an arbitrary crashing protocol, and let $[\mathbf{N}, \mathbf{L}]$ be any possible state of $A(U)$ for a graph $G = (V, E)$. Then*:

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

PROOF. We use the Unreliable FIFO Node State Generation Lemma, Lemma 8.11, to show that there exists a node state generator $\mathbf{G_N}$ for possible node state vector $\mathbf{N}$, where $\mathbf{G_N}$ is a concatenation of send sequences for $A(\mathrm{U})$.

We use the Unreliable FIFO Link State Generation Lemma, Lemma 8.12, to show that there exists a link state generator $\mathbf{G_L}$ for possible link state vector $\mathbf{L}$, where $\mathbf{G_L}$ is a concatenation of send sequences for $A(\mathrm{U})$.

Using the Unreliable FIFO Concatenation Lemma, Lemma 8.10, we see (because both generators are concatenations of send sequences) that:

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{G_N G_L}].$$

Using the Unreliable FIFO playout lemma, Lemma 8.13, we see that

$$[\mathbf{N_0}, \mathbf{G_N G_L}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

The theorem now follows from the last two transitions and the transitivity of transitions. □

9. *Proofs for Reliable FIFO*

In this section, we present the proofs of the result for reliable FIFO links. The proofs for reliable FIFO and reliable non-FIFO are deliberately structured to be similar to the proof for unreliable links (see previous section), except for some important differences that we will highlight. These differences occur because of the inability to directly lose packets. However, at the highest level the proof strategy is still to: (1) use a similar concatenation construction to load the links with an arbitrary concatenation of send sequences and (2) playout the chosen sequences to produce arbitrary node states. The playout construction is very different for the three link models.

We start by stating the Reliable FIFO loss lemma and proving it. This captures the limited ability of the CAM model to lose packets. Note that as the links are reliable, this lemma actually expresses the ability of the system to lose packets from a link by receiving the packets and crashing the receiving node. We do not state or prove the Crash lemma, Lemma 8.1, which is the same for all link models.

LEMMA 9.1 (RELIABLE FIFO LOSS). *Consider a protocol $A(R)$ for a graph $G = (V, E)$. Let $\mathbf{L}$ be a link state vector such that for some $(i, j) \in E$, $\mathbf{L}[i, j] = P\,Q$, where P, Q are finite sequences of packets. Then for all node vectors $\mathbf{N}$, $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}', \mathbf{L}']$, where $\mathbf{L}' = \mathbf{L}$ except that $\mathbf{L}'[i, j] = Q$ and $\mathbf{N}'$ is the same as $\mathbf{N}$ except that $\mathbf{N}'[j]$ is the unique start state of node j.*

PROOF. Intuitively, this lemma is proved by repeatedly receiving a packet on link (i, j) and crashing the node j until the sequence P has been "lost". We will prove that if there exists a finite execution α of $A(\mathrm{R})$ with schedule β (say), such that the final state of α is $[\mathbf{N}, \mathbf{L}]$ then there exists a finite extension α' of α with schedule $\beta\beta'$, such that the final state of α' is $[\mathbf{N}, \mathbf{L}']$. If $P = p_1 p_2 \cdots p_l$ and $Q = q_1 q_2 \cdots q_m$, then we extend the original schedule with $\beta' = crash^j receive^{i,j}(p_1) crash^j receive^{i,j}(p_2) \cdots receive^{i,j}(p_l) crash^j$. It is not hard to verify that this is a valid schedule and results in removing all the packets in P from link (i, j) while leaving node j in its unique start state. All other node and link states remain unchanged. □

Next, we prove the reliable FIFO Rotate Node lemma which states (exactly as in the CAML model) that if a node j is supplied with all the packets it receives in some execution α, then node j can go to its corresponding state at the end of α and can be made to send all the packets it sends in α. All other packets in the link (formalized by a link state vector $\mathbf{O}$), remain in the links, except that the $\mathbf{O}$ packets "rotate" to the head of each link, and the sequence sent by j rotates to the tail of the links outgoing from node j. This lemma once again expresses the effect of asynchrony and locality in the context of reliable FIFO links. For the

statement of the lemma, recall that $\mathbf{N_0}$ is the node state vector such that for every $i \in V$, $\mathbf{N_0}[i]$ is the unique start state of node A^i.

LEMMA 9.2 (RELIABLE FIFO ROTATE NODE). *Consider a finite execution α of $A(R)$. For all node state vectors* **N** *and link state vectors* **O** *of $A(R)$ and for any $j \in V$,*

$$[\mathbf{N}, \mathbf{RCV}^j(\alpha)\mathbf{O}] \overset{\beta}{\rightsquigarrow} [\mathbf{N'}, \mathbf{O}\ \mathbf{SND}^j(\alpha)],$$

where $\beta = crash^j\ acts^j(\alpha)$ and **N**′ *is the same as* **N** *except that $\mathbf{N}'[j] = last\_state^j(\alpha)$.*

PROOF. Identical to proof of Lemma 9.2 except that we replace unreliable link channel automata (U) with reliable link automata (R). □

The next corollary states that by loading the links with the receive sequences for all nodes in some execution α, we can cause every node j to emit the sequence of packets that j emits in α. Once again, the old packets remain unchanged. It is again identical to the corresponding lemma for unreliable link systems (compare with Corollary 8.4).

COROLLARY 9.3 (RELIABLE FIFO ROTATE). *Consider a finite execution α of $A(R)$. For all node state vectors* **N** *and link state vectors* **O** *of $A(R)$,*

$$[\mathbf{N}, \mathbf{RCV}(\alpha)\mathbf{O}] \rightsquigarrow [\mathbf{N'}, \mathbf{O}\ \mathbf{SND}(\alpha)],$$

where $\forall j \in V$, $\mathbf{N}'[j] = last\_state^j(\alpha)$.

PROOF. This corollary is proved by applying the Rotate Node lemma (Lemma 9.2) at each node $j \in V$. □

The next lemma shows that we can "pump" up the send sequence (on all links) corresponding to a prefix α_k of execution α, and produce a send sequence corresponding to a longer prefix α_{k+1}. Once again, old packets "rotate" to the front of the links. This lemma is rather different from the corresponding lemma for unreliable links. Compare Lemma 8.6 and Lemma 9.4. A major difference is that we need to drive all nodes to initial states in order to pump up the send sequence; this was not necessary for the unreliable link model, in which we can lose packets directly without crashing nodes.

LEMMA 9.4 (RELIABLE FIFO ROTATE WITH INCREMENT). *Consider an execution α of $A(R)$ of length n. For all node state vectors* **N** *and link state vectors* **O** *and for $k < n$,*

(*a*) $[\mathbf{N}, \mathbf{SND}(\alpha_k)\ \mathbf{O}] \rightsquigarrow [\mathbf{N_0}, \mathbf{O}\ \mathbf{SND}(\alpha_{k+1})]$
(*b*) $[\mathbf{N}, \mathbf{SND}(\alpha_k)\ \mathbf{O}] \rightsquigarrow [\mathbf{N_0}, \mathbf{O}\ \mathbf{SND}(\alpha_k)]$

PROOF. For every link $U^{i,j}$, $(i, j) \in E$, $rcv^{i,j}(\alpha_{k+1})$ is a prefix of $snd^{i,j}(\alpha_k)$. This follows because the link is reliable and FIFO and all packets which are received on a link in an execution must be sent on the link before the last receive. Thus $\mathbf{SND}(\alpha_k)\ \mathbf{O}$ can be written as $\mathbf{RCV}(\alpha_{k+1})\ \mathbf{O'}\ \mathbf{O}$. By the Rotate corollary (Corollary 9.3), we get,

$$[\mathbf{N}, \mathbf{RCV}(\alpha_{k+1})\mathbf{O'}\ \mathbf{O}] \rightsquigarrow [\mathbf{N'}, \mathbf{O'}\ \mathbf{O}\ \mathbf{SND}(\alpha_{k+1})],$$

where for all $i \in V$, $\mathbf{N}'[i] = last\_state^i(\alpha_{k+1})$. By applying the Reliable FIFO loss lemma (Lemma 9.1), to each link we get

$$[\mathbf{N}', O' \text{ O } \text{SND}(\alpha_{k+1})] \rightsquigarrow [\mathbf{N_0}, \mathbf{O\ SND}(\alpha_{\mathbf{k+1}})].$$

Thus, part (a) of the lemma is proved by transitivity of transitions. Part (b) is proved similarly as $\mathbf{RCV}(\alpha_k)$ is a prefix of $\mathbf{RCV}(\alpha_{k+1})$. □

For the next lemma, we again define a link state vector **L** to be a *concatenation of send sequences* for $A(\text{R})$, if there exist finite executions $\alpha^1, \alpha^2, \ldots, \alpha^m$ of $A(\text{R})$ such that: $\mathbf{L} = \mathbf{SND}(\alpha^1)\mathbf{SND}(\alpha^2) \cdots \mathbf{SND}(\alpha^m)$.

Our next lemma states that any send sequence that is sandwiched in between two link state vectors **O** and **O′** can be rotated to the leftmost position as long as both **O** and **O′** are a concatenation of send sequences. The proof is similar to that of the corresponding lemma for unreliable links (Lemma 8.8).

LEMMA 9.5 (RELIABLE FIFO ANY ROTATE). *Let* **O** *and* **O′** *each be a concatenation of send sequences for* $A(R)$. *Let* α *be a finite execution of* $A(R)$. *Let* $\mathbf{L} = \mathbf{O}$ $\mathbf{SND}(\alpha)$ $\mathbf{O}'$ *and* $\mathbf{L}' = \mathbf{SND}(\alpha)$ $\mathbf{O}'$ $\mathbf{O}$. *Then*:

$$[\mathbf{N_0}, \mathbf{L}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}'].$$

PROOF. We start by rewriting **O** and **O′** as a concatenation of send sequences (possibly from different executions.) We are then left with a link state vector that is a concatenation of send sequences which contains $\mathbf{SND}(\alpha)$. To rotate a concatenation of send sequences one step to the left (so that the leftmost send sequence moves to the right) we simply apply the Reliable FIFO Rotate with Increment lemma (Lemma 9.4(b)) to the leftmost send sequence (which is say $\mathbf{SND}(\alpha')$). We keep rotating one step at a time until $\mathbf{SND}(\alpha)$ is in the leftmost position. □

Consider a send sequence $\mathbf{SND}(\alpha^k)$ corresponding to a prefix of an execution α that is sandwiched in between two link state vectors **O** and **O′**. Suppose both **O** and **O′** are a concatenation of send sequences. Then $\mathbf{SND}(\alpha^k)$ can be pumped up to $\mathbf{SND}(\alpha^{k+1})$ without changing **O** and **O′**, or their relative positions. The lemma and proof are essentially similar to the corresponding lemma for unreliable links (Lemma 8.9), except that we use the corresponding lemmas for reliable links in the proof below.

LEMMA 9.6 (RELIABLE FIFO INCREMENT). *Let* **O** *and* **O′** *each be a concatenation of send sequences for* $A(R)$. *Let* α *be a finite execution of* $A(R)$ *of length* n. *Let* $\mathbf{L} = \mathbf{O}$ $\mathbf{SND}(\alpha_k)$ $\mathbf{O}'$ *for* $0 \leq k < n$. *Let* $\mathbf{L}' = \mathbf{O}$ $\mathbf{SND}(\alpha_{k+1})$ $\mathbf{O}'$. *Then*

$$[\mathbf{N_0}, \mathbf{L}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}'].$$

PROOF. The proof proceeds in three steps. We first rotate **L**, to bring $\mathbf{SND}(\alpha_k)$ to the head. Then, we rotate and increment $\mathbf{SND}(\alpha_k)$ to $\mathbf{SND}(\alpha_{k+1})$ and crash each node to bring the nodes to the initial states. Finally, we rotate back $\mathbf{SND}(\alpha_{k+1})$ to the original position. The first step uses the Reliable FIFO Any Rotate lemma (Lemma 9.5). The second step uses the Reliable FIFO Rotate with Increment lemma (Lemma 9.4), and the third step uses the Reliable FIFO Any Rotate lemma (Lemma 9.5) again. □

LEMMA 9.7 (RELIABLE FIFO CONCATENATION). *Let* $\mathbf{L}$ *be any concatenation of send sequences for* $A(R)$. *Then*

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}].$$

PROOF. Identical to the proof of Lemma 8.10 except that we use the corresponding lemmas for the CAM model. Thus we use the Reliable FIFO Rotate Corollary (Corollary 9.3) in place of the Unreliable FIFO Rotate Corollary (Corollary 8.4). Similarly, we use the Reliable FIFO Increment Lemma (Lemma 9.6) in place of the Unreliable FIFO Increment Lemma (Lemma 8.9). □

For the main theorem, we assume that the definition of a *node state generator* is the same as that for unreliable links. We now describe the playout lemma for Reliable FIFO case where the packets in a node state generator are played out to drive the nodes to any possible state. Notice that we do not need a link state generator for the CAM system as we only aim to control node states. The playout constructions we now describe are different from the ones used for the CAML system.

LEMMA 9.8 (RELIABLE FIFO PLAYOUT). *There exists some node state generator* $\mathbf{G}$ *for any possible node state vector* $\mathbf{N}$ *such that* $\mathbf{G}$ *is a concatenation of send sequences. Also, given any node state vector* $\mathbf{N}'$,

$$[\mathbf{N}', \mathbf{G}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

where $\mathbf{L}$ *is some link state vector.*

PROOF. If $\mathbf{N}$ is a possible node state vector, then for all $i \in V$, $\mathbf{N}[i] = last\_state^i(\alpha^i)$, for some execution α^i of $A(\mathrm{R})$. Thus, we set $\mathbf{G} = \Pi_{\forall i \in V}\mathbf{SND}(\alpha^i)$. This proves the first part of the lemma.

To prove the second part, we use the fact that $\mathbf{G} = \Pi_{\forall i \in V}\mathbf{SND}(\alpha^i)$ for some set of executions α^i such that $\forall i \in V$, $\mathbf{N}[i] = last\_state^i(\alpha^i)$. Note that for any link (i, j), $\mathbf{G}[i, j] = \mathbf{T}[i, j]\ rcv^{i,j}(\alpha^j)\ \mathbf{T}'[i, j]$, where $\mathbf{T}$, $\mathbf{T}'$ are some link state vectors. The above follows from the fact that $rcv^{i,j}(\alpha^j)$ is a prefix of $snd^{i,j}(\alpha^j)$. By applying the Reliable FIFO link loss lemma, Lemma 9.1, for each link we get,

$$[\mathbf{N}', \mathbf{G}] \rightsquigarrow [\mathbf{N_0}, \mathbf{G}'],$$

where $\mathbf{G}'$ is a link state vector such that $\mathbf{G}'[i, j] = rcv^{i,j}(\alpha^j)\ \mathbf{T}'[i, j]$. In other words, $\mathbf{G}' = \Pi_{\forall j \in V}\mathbf{RCV}^j(\alpha^j)\ \mathbf{T}'$ We finally prove the lemma by applying the Reliable FIFO Rotate Node lemma, Lemma 9.2, at each node j. □

We are now ready to prove our main theorem for reliable links, which states that we can drive the system to any possible node state. The proof uses the earlier concatenation construction to load the links with a node generator for the goal state; it then uses the previous lemma to play out the node generator to result in the goal state.

THEOREM 9.9 (RELIABLE FIFO ANY NODE STATE). *Let* A *be an arbitrary crashing protocol, and let* $\mathbf{N}$ *be any possible node state vector of* $A(R)$ *for a graph* $G = (V, E)$. *Then there exists a state* $[\mathbf{N}, \mathbf{L}]$ *of* $A(R)$ *such that*

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

PROOF. From the Reliable FIFO Playout lemma (Lemma 9.8), we know that for the given node state vector **N** there exists a node state generator **G** which is a concatenation of send sequences. Since **G** is a concatenation of send sequences, we have from the Reliable FIFO Concatenation lemma (Lemma 9.7):

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{G}].$$

Since **G** is a generator for node state vector **N**, we have from the Playout lemma (Lemma 9.8),

$$[\mathbf{N_0}, \mathbf{G}] \rightsquigarrow [\mathbf{N}, \mathbf{L}],$$

where **L** is some link state vector. From the transitivity of transitions, we get,

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N}, \mathbf{L}]. \qquad \square$$

10. *Proofs for Reliable non-FIFO*

In this section, we present the proofs of the result for reliable non-FIFO links. As the links are non-FIFO, their state is expressed by a set of packets and not a sequence of packets. However, for uniformity, we will retain the notation used in the last two sections. We only note that packets can be in any order in the link, and that the concatenation of two sets of packets in the links is the same as their union. However, the definitions for receive and send sequences stand unchanged, as they are concerned with the *sequence* of packets sent or received in some execution and not with the state of the link itself.

As the proofs of the lemmas needed for the concatenation construction are identical to those for reliable FIFO links (except with the non-FIFO link automaton RN substituted for the FIFO link automaton R), we shall state the final loss, concatenation, and rotate node lemmas for non-FIFO links without proof.

The only additional delicate point for non-FIFO links is that we cannot deliver packets without respecting the ordering of packet tags (see model of a live link in Section 3). However, we can easily get around this by finding some initial setting of the *tags* array that makes any packet delivery order possible [Jayaram 1996]. Thus, if we start with state $[\mathbf{N}, \mathbf{L}]$ and if by delivering some packet on link (i, j) to node j we result in state $[\mathbf{N}', \mathbf{L}']$, it is indeed true that $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}', \mathbf{L}']$. We will assume this fact implicitly in what follows.

LEMMA 10.1 (RELIABLE NON-FIFO LOSS). *Consider a protocol $A(RN)$ for a graph $G = (V, E)$. Let* **L** *be a link state vector such that for some $(i, j) \in E$, $\mathbf{L}[i, j] = O\ Q$, where O, Q are finite sets of packets. Then for all node vectors* **N**, $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}', \mathbf{L}']$, *where* $\mathbf{L}' = \mathbf{L}$ *except that* $\mathbf{L}'[i, j] = Q$ *and* $\mathbf{N}'$ *is the same as* **N** *except that* $\mathbf{N}'[j] =$ *the unique start state.*

PROOF. Similar to the proof of Lemma 9.1. $\square$

LEMMA 10.2 (RELIABLE NON-FIFO ROTATE NODE). *Consider a finite execution α of $A(RN)$. For all node state vectors* **N** *and link state vectors* **O** *of $A(RN)$ and for any $j \in V$,*

$$[\mathbf{N}, \mathbf{RCV}^j(\alpha)\mathbf{O}] \overset{\beta}{\rightsquigarrow} [\mathbf{N}', \mathbf{O}\ \mathbf{SND}^j(\alpha)],$$

where $\beta = crash^j\ acts^j(\alpha)$ *and* $\mathbf{N}'$ *is the same as* $\mathbf{N}$ *except that* $\mathbf{N}'[j] = last\_state^j(\alpha)$.

PROOF. Similar to the proof of Lemma 9.2. □

LEMMA 10.3 (RELIABLE NON-FIFO CONCAT). *Let* $\mathbf{L}$ *be any concatenation of send sequences for* $A(RN)$. *Then*:

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{L}].$$

PROOF. Identical to the proof of Lemma 9.7 using exactly the same supporting lemmas, except with RN substituted for R in the proof and all supporting lemmas. □

The statement and proof of the concatenation lemma for reliable non-FIFO is essentially identical to that of the reliable FIFO case. However the following lemmas and proofs which describe the playout construction for reliable non-FIFO links are very different. Before we proceed, we need some new definitions and notation.

The result for the unreliable FIFO case showed that it is possible to take the state of the system to any possible node state vector and link state vector. Correspondingly, it was necessary to load the links with the node state generator and the link state generator. Similarly, the result for the reliable FIFO case showed that it is possible to take the state of the system to any possible node state vector and to do that it was necessary to load the links with the node state generator. The result for the reliable non-FIFO case states that it is possible to control any acyclic subset of the links and nodes.

Correspondingly, we need to load the links with an *acyclic state generator* that consists of the concatenation of a set of send sequences that can drive the acyclic subset of the graph to the desired goal state. Let us first define an acyclic state generator. The definition of an acyclic state generator is similar to that of a link or node state generator; the only difference is that it only applies to those nodes or links which are in the acyclic subset. Essentially, an acyclic state generator has all the packets needed to drive the acyclic subset to its goal state.

Consider a graph $G = (V, E)$ and a set of edges and nodes (V_a, E_a) where $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle of links and nodes wholly contained in $V_a \cup E_a$. Consider an assignment $\mathcal{A}$ of a possible node state $s^i = last\_state^i(\alpha^i)$ to every node $i \in V_a$ (where α^i is some execution of $A(\mathrm{RN})$) together with an assignment of a possible link state to each link $(i, j) \in E_a$. Let $\mathbf{G_N} = \Pi_{\forall i \in V_a}\mathbf{SND}(\alpha^i)$. If the state of link (i, j) is $p_1 p_2 \cdots p_n$, then $\forall k$, $p_k \in snd^{i,j}(\alpha_k^{i,j})$ for some execution $\alpha_k^{i,j}$ of $A(\mathrm{RN})$. Let $\mathbf{G_{i,j}} = \Pi_{\forall k}\mathbf{SND}(\alpha_k^{i,j})$. Let us call $\mathbf{G_L} = \Pi_{\forall (i,j) \in E_a}\mathbf{G_{i,j}}$. Finally, we call $\mathbf{G} = \mathbf{G_N}\mathbf{G_L}$ the *acyclic state generator* corresponding to the acyclic state assignment $\mathcal{A}$.

Given the acyclic state generator, we need to describe the playout strategy that plays out the packets in the generator to achieve the desired possible acyclic state. While the playout strategy is formalized in the proof of the playout lemma described later, it helps to first intuitively understand the strategy.

If link $(i, j) \in E_a$ and node $j \in V_a$, then we can think of node j as being dependent on link (i, j) in the following sense. When we drive link (i, j) to its goal state, we may have to lose packets, which can only be achieved by crashing node j. Thus, it makes sense to drive node j to its final state *after* driving link

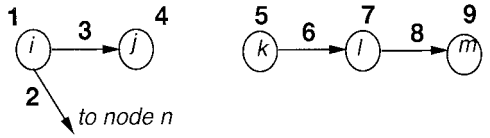

FIG. 18. Assigning a *height* numbering to each node and link in the acyclic subset. The figure shows two connected components. Notice that the heights respect the intuitive dependency relation, and that all heights in the second connected component are greater than the maximum height in the first connected component.

(i, j) to its final state.[7] Similarly, if node $j \in V_a$ and link $(j, i) \in E_a$, then driving node j to its desired goal state may result in j emitting packets that are not part of the final goal state for link (j, i). Thus it makes sense to drive (j, i) to its final goal state *after* driving node j to its final state, and we can think of link (j, i) as being dependent on node j.

The preceding paragraph shows that we can define a dependency relation between elements (i.e., nodes or links) in the acyclic subset (V_a, E_a). We will use this dependency relation to determine the order in which we will playout packets and drive individual components to their final states. We formalize the dependency relation using a *height* function that assigns an integer to each component such that $height(x) < height(y)$ if y depends on x.

To allow a simple inductive proof, we will assign each component a unique *height*. This can easily be done by topologically sorting and numbering each connected component in (V_a, E_a). We then add the maximum value of *height* in the ith subcomponent to each number in the $i + 1$th subcomponent. This is illustrated in Figure 18 where $V_a = \{i, j, k, l, m\}$ and $E_a = \{(i, n), (i, j), (k, l), (l, m)\}$. Notice that there are no cycles in this set of nodes and links, and that node n is not in the subset, although link (i, n) is.

Notice also that there are two connected components. The *height* of each element (node or link) is shown by the bolded number above the node or link. We make sure that each node has a *height* that is less than any of its outgoing links, and that each node has a *height* greater than any of its incident links. After numbering the first component, we start numbering the second component where we left off in the first component (Figure 18).

We now formally define the *height* function:

LEMMA 10.4 (RELIABLE NON-FIFO HEIGHT). *Consider a graph $G = (V, E)$ and a set of edges and nodes (V_a, E_a) where $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle of links and nodes wholly contained in $V_a \cup E_a$, there exists a function height on elements of $V \cup E$ such that*:

(a) $x \notin V_a \cup E_a$, $height(x) = undefined$
(b) $x \in V_a \cup E_a$, $0 \leq height(x) < |V_a \cup E_a|$
(c) $x, y \in V_a \cup E_a$, $height(x) = height(y) \rightarrow x = y$
(d) $x, y \in V_a \cup E_a$, *and there is a path from x to y wholly contained in* $V_a \cup E_a$, *then* $height(x) < height(y)$.

[7] Although we say that we are driving a node or a link to a state, it should be clear that we are really driving the corresponding node or link *automaton* to the desired state.

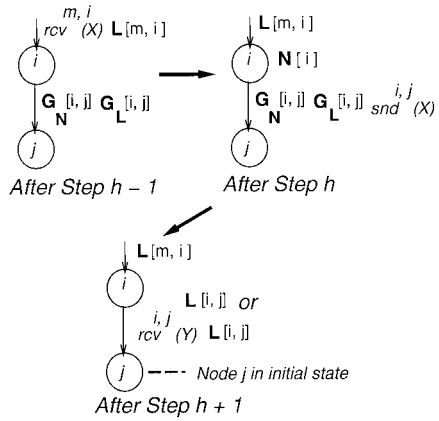

FIG. 19. Illustrating the inductive construction for reaching a possible acyclic state. If $height(i) = h$, in Step h, we allow node i to receive all the packets it would have received in some execution $X = \alpha_i$ and go into its desired final state $\mathbf{N}[i]$ while emitting $snd^{i,j}(X)$ on link (i, j). If $height(i, j) = h + 1$, in Step $h + 1$, we lose packets on link (i, j) to drive (i, j) to its final state $\mathbf{L}[i, j]$. If j is in the acyclic component, we must also leave $rcv^{i,j}(Y)$, the packets required on link (i, j) to drive node j to its final state in execution $Y = \alpha_j$.

PROOF. We know that as there is no cycle among components of (V_a, E_a), so $V_a \cup E_a$ consists of a forest of connected acyclic subcomponents of $G = (V, E)$. We can give a proof of the existence of the function *height* by induction on the number of acyclic connected subcomponents. For the base case of no acyclic connected subcomponents (i.e., when $V_a \cup E_a$ is empty), the proof is trivial as *height* is then undefined for all components of the graph G. Assume that there exists a function *height* for a forest of i connected subcomponents that satisfies the above properties. To extend *height* to the $i + 1$st connected subcomponent, all we have to do is to number the $i + 1$st connected subcomponent topologically from 1 to the number of components in it. Then by adding the maximum value of *height* in the numbering till the ith subcomponent to each number in the $i + 1$th subcomponent we get an extension of the function *height* to the $i + 1$th subcomponent as well. Moreover, this extended function height satisfies the properties as well because the $i + 1$th subcomponent is acyclic and because of the properties of topological ordering. □

Our strategy will be to process element $x \in V_a \cup E_a$ in Step h if $height(x) = h$ using an inductive construction illustrated in Figure 19. Assume $height(i)$ is h and $height(i, j)$ is $h + 1$. In order to ensure that node i can be processed properly, we must ensure that at the end of Step $h - 1$, the state of the system is as shown on the top left of Figure 19. Notice that link (m, i) (node m is not shown) has the set $rcv^{m,i}(X)L[m, i]$. $L[m, i]$ is the desired goal state of link (m, i). Also, $rcv^{m,i}(X)$ is the packets that i must receive from link (m, i) in order to reach its state at the end of execution $X = \alpha_i$. Notice also that link (i, j) is in its initial state corresponding to the concatenation of the node and link state generators $\mathbf{G_N}$ and $\mathbf{G_L}$.

In Step h, we allow node i to receive all the packets it would have received in execution $X = \alpha_i$ and go into its desired final state $\mathbf{N}[i] = last\_state(\alpha_i)$ while emitting $snd^{i,j}(X)$ on link (i, j). The resulting state is shown on the top right of Figure 19. In Step $h + 1$, we process link (i, j). There are two cases.

If node j is not in the acyclic component, we do not have to worry about driving node j to a desired state. Thus, we can simply lose enough packets to arrive at the desired final state for link (i, j) which is $\mathbf{L}[i, j]$. This can be easily done because $\mathbf{G_L}$ was constructed so that $\mathbf{L}[i, j]$ is a subsequence of $\mathbf{G_L}[i, j]$. The loss is effected by having node j receive each unwanted packet (recall we are dealing with non-FIFO links) and then crashing node j.

The case when node j is part of the acyclic component is similar except that we must also leave the sequence $rcv^{i,j}(Y)$ on link (i, j) in addition to $\mathbf{G_L}[i, j]$. $rcv^{i,j}(Y)$ is needed in order to drive node j to *last_state*(α_j), where $\alpha_j = Y$. Once again this is easy to do because done because $\mathbf{G_N}$ was constructed so that $rcv^{i,j}(\alpha_j)$ is a subsequence of $\mathbf{G_N}[i, j]$. This final state (with its two cases) is shown at the bottom of Figure 19.

Thus, the construction proceeds by driving nodes and links into their final states in *height* order. In case of links (e.g., (i, j) in Figure 19), we will sometimes also leave some additional packets required to drive the receiver end of the link to its final state. However, these additional packets will be removed when the receiver end of the link (e.g., j in Figure 19) is driven to its final state.

We proceed to formalize the process described in Figure 19. We exhibit a proof of the playout lemma for the reliable non-FIFO case by induction on *height*. For the sake of convenience, we shall first describe the intermediate states in the inductive construction.

Let $hmax = |V_a \cup E_a|$. Let A be an arbitrary crashing protocol, and consider $A(\mathrm{RN})$ for a graph $G = (V, E)$. Consider (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle wholly among elements of $V_a \cup E_a$. Consider also an assignment $\mathcal{A}$ of any possible node state s^i for every node $i \in V_a$ and of any possible link state $s^{i,j}$ for every link $(i, j) \in E_a$. Consider a state $[\mathbf{N}, \mathbf{L}]$ of $A(\mathrm{RN})$ such that $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i, j] = s^{i,j}$, $(i, j) \in E_a$.

Given a state $[\mathbf{N}, \mathbf{L}]$ defined as above, a function *height* with the properties stated in Lemma 10.4, then for $0 \leq h \leq hmax$ we define $[\mathbf{N^h}, \mathbf{L^h}]$ as a state such that:

—If $i \in V_a$, $height(i) < h$, then $\mathbf{N^h}[i] = \mathbf{N}[i]$

—If $(i, j) \in E_a$, $height(i, j) < h$, $j \in V_a$, $height(j) < h$, then $\mathbf{L^h}[i, j] = \mathbf{L}[i, j]$

—Else if $(i, j) \in E_a$, $height(i, j) < h$, $j \in V_a$, $height(j) \geq h$, then $\mathbf{L^h}[i, j] = rcv^{i,j}(\alpha^j)\ \mathbf{L}[i, j]$, where $last\_state^j(\alpha^j) = \mathbf{N}[j]$

—Else if $(i, j) \in E_a$, $height(i, j) < h$, $j \notin V_a$, then $\mathbf{L^h}[i, j] = \mathbf{L}[i, j]$

—Else if $(i, j) \in E_a$, $height(i, j) \geq h$, then $\mathbf{L^h}[i, j] = O\ \mathbf{G_N}[i, j]\ \mathbf{G_L}[i, j]$, where O is some sequence of packets.

—Else if $(i, j) \notin E_a$, $j \in V_a$, $height(j) \geq h$, then $\mathbf{L^h}[i, j] = O\ \mathbf{G_N}[i, j]\ \mathbf{G_L}[i, j]$, where O is some sequence of packets.

Figure 19 provides some intuition for the form of the definition of $[\mathbf{N^h}, \mathbf{L^h}]$, which represents the state after Step $h - 1$.

To prove the playout lemma, we first show that $[\mathbf{N^0}, \mathbf{L^0}]$ is indeed the initial state after concatenation: $[\mathbf{N_0}, \mathbf{G_N}\ \mathbf{G_L}]$. This is the base case of the induction. We then show that the result of the induction, $[\mathbf{N^{hmax}}, \mathbf{L^{hmax}}]$ is indeed the desired final state $[\mathbf{N}, \mathbf{L}]$. Finally, we prove the inductive step and show how to go from $[\mathbf{N^h}, \mathbf{L^h}]$ to $[\mathbf{N^{h+1}}, \mathbf{L^{h+1}}]$. We start with the first step.

LEMMA 10.5 (RELIABLE NON-FIFO START). *Let A be an arbitrary crashing protocol, and consider $A(RN)$ for a graph $G = (V, E)$. Consider (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle wholly among elements of $V_a \cup E_a$. Consider also an assignment $\mathcal{A}$ of any possible node state s^i for every node $i \in V_a$ and of any possible link state $s^{i,j}$ for every link $(i,j) \in E_a$. Consider a state $[\mathbf{N}, \mathbf{L}]$ of $A(RN)$ such that $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i,j] = s^{i,j}$, $(i,j) \in E_a$. Then,*

$$[\mathbf{N_0}, \mathbf{G_N}\ \mathbf{G_L}] = [\mathbf{N^0}, \mathbf{L^0}].$$

PROOF. For all $x \in V_a \cup E_a$, $height(x) \geq 0$. Thus, the lemma is proved trivially because for all $(i,j) \in E$ each link state is $\mathbf{G_N}[i,j]\ \mathbf{G_L}[i,j] = \mathbf{L^0}[i,j]$. □

We now show that the result of the induction, $[\mathbf{N^{hmax}}, \mathbf{L^{max}}]$ is indeed the desired final state $[\mathbf{N}, \mathbf{L}]$.

LEMMA 10.6 (RELIABLE NON-FIFO LAST). *Let A be an arbitrary crashing protocol, and consider $A(RN)$ for a graph $G = (V, E)$. Consider (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle wholly among elements of $V_a \cup E_a$. Let $hmax = |V_a \cup E_a|$. Consider also an assignment of any possible node state s^i for every node $i \in V_a$ and of any possible link state $s^{i,j}$ for every link $(i,j) \in E_a$. Consider a state $[\mathbf{N}, \mathbf{L}]$ of $A(RN)$ such that $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i,j] = s^{i,j}$, $(i,j) \in E_a$. Then,*

$$[\mathbf{N}, \mathbf{L}] = [\mathbf{N^{hmax}}, \mathbf{L^{hmax}}].$$

PROOF. For all $x \in V_a \cup E_a$, $height(x) < hmax$. Thus, for all $i \in V_a$, $\mathbf{N^{hmax}}[i] = \mathbf{N}[i]$ and for all $(i,j) \in E_a$, $\mathbf{L^{hmax}}[i,j] = \mathbf{L}[i,j]$. □

We are now ready to prove the inductive step and show how to go from $[\mathbf{N^h}, \mathbf{L^h}]$ to $[\mathbf{N^{h+1}}, \mathbf{L^{h+1}}]$.

LEMMA 10.7 (RELIABLE NON-FIFO MIDDLE). *Let A be an arbitrary crashing protocol, and consider $A(RN)$ for a graph $G = (V, E)$. Consider (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle wholly among elements of $V_a \cup E_a$. Let $hmax = |V_a \cup E_a|$. Consider also an assignment of any possible node state s^i for every node $i \in V_a$ and of any possible link state $s^{i,j}$ for every link $(i,j) \in E_a$. Consider a state $[\mathbf{N}, \mathbf{L}]$ of $A(RN)$ such that $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i,j] = s^{i,j}$, $(i,j) \in E_a$. Then, for $0 \leq h < hmax$,*

$$[\mathbf{N^h}, \mathbf{L^h}] \rightsquigarrow [\mathbf{N^{h+1}}, \mathbf{L^{h+1}}].$$

PROOF. Consider the element $x \in V_a \cup E_a$ such that $height(x) = h$. There are two cases, depending on whether x is a node or a link. Refer to Figure 19 for the main idea of the proof.

Case 1, *x is a node*. If $x = i \in V_a$ then by the property of *height*, for all $(m,i) \in E_a$, $height(m,i) < h$, and so for such links (m,i), $\mathbf{L^h}[m,i] = rcv^{m,i}(\alpha^i)\ \mathbf{L}[m,i]$ where $last\_state^i(\alpha^i) = \mathbf{N}[i]$. For all links $(m,i) \notin E_a$, $\mathbf{L^h}[m,i] = O\ \mathbf{G_N}[m,i]\ \mathbf{G_L}[m,i] = rcv^{m,i}(\alpha^i)\ O'$, where $\mathbf{G_N}\ \mathbf{G_L}$ is the generator for the acyclic state and O, O' are some sequences of packets. Thus, we can apply the Reliable non-FIFO Rotate Node lemma (Lemma 10.2) at node i.

After applying the lemma, the state of node i becomes $last\_state^i(\alpha^i) = \mathbf{N^{h+1}}[i]$. Also applying the Rotate Node lemma adds $snd^{i,k}(\alpha^i)$ to every link

(i, k). However, for $(i, k) \in E_a$, $height(i, k) \geq h + 1$ the state of all links (i, k) becomes $O\ \mathbf{G_N}[i, k]\ \mathbf{G_L}[i, k] = \mathbf{L^{h+1}}[i, k]$ where O is some sequence of packets. The state of all links $(m, i) \in E_a$ after applying the lemma is $\mathbf{L}[m, i] = \mathbf{L^{h+1}}[m, i]$. All other links and nodes in $[\mathbf{N^h}, \mathbf{L^h}]$ have the same state as in $[\mathbf{N^{h+1}}, \mathbf{L^{h+1}}]$. Thus the lemma is proved when x is a node.

Case 2, *x is a link*. Suppose x is a link (i, j) such that $height(i, j) = h$. Then $\mathbf{L^h}[i, j] = O\ \mathbf{G_N}[i, j]\ \mathbf{G_L}[i, j]$. Consider the node j. If $j \in V_a$, then $height(j) \geq h + 1$. $O\ \mathbf{G_N}[i, j]\ \mathbf{G_L}[i, j]$ contains $rcv^{i,j}(\alpha^j)\ \mathbf{L}[i, j]$ where $last\_state^j(\alpha^j) = \mathbf{N}[j]$. Thus, by applying the Reliable non-FIFO Loss lemma (Lemma 10.1), we can lose packets so that the state of the link (i, j) becomes $rcv^{i,j}(\alpha^j)\ \mathbf{L}[i, j] = \mathbf{L^{h+1}}[i, j]$. If $j \notin V_a$, then, by applying the same Loss lemma, we can lose packets to make the state of link (i, j) to be $\mathbf{L}[i, j] = \mathbf{L^{h+1}}[i, j]$. All other links and nodes in $[\mathbf{N^h}, \mathbf{L^h}]$ have the same state as in $[\mathbf{N^{h+1}}, \mathbf{L^{h+1}}]$. Thus, the lemma is also proved for the case when x is a link. □

We can now put together the last three lemmas to obtain an inductive proof of the playout lemma for non-FIFO links.

LEMMA 10.8 (RELIABLE NON-FIFO PLAYOUT). *Let A be an arbitrary crashing protocol, and consider A(RN) for a graph* $G = (V, E)$. *Consider* (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ *such that there is no cycle wholly among elements of* $V_a \cup E_a$. *Let* $hmax = |V_a \cup E_a|$. *Consider also an assignment* $\mathcal{A}$ *of any possible node state* s^i *for every node* $i \in V_a$ *and of any possible link state* $s^{i,j}$ *for every link* $(i, j) \in E_a$. *Consider a state* $[\mathbf{N}, \mathbf{L}]$ *of A(RN) such that* $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i, j] = s^{i,j}$, $(i, j) \in E_a$. *Let* $\mathbf{G_N}\ \mathbf{G_L}$ *be the acyclic state generator corresponding to the acyclic state assignment* $\mathcal{A}$ *such that* $\mathbf{G_N}\ \mathbf{G_L}$ *is a concatenation of send sequences. Then*,

$$[\mathbf{N_0}, \mathbf{G_N}\ \mathbf{G_L}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

PROOF. From the Reliable non-FIFO middle lemma (Lemma 10.7), we know that for $0 \leq h < hmax$,

$$[\mathbf{N^h}, \mathbf{L^h}] \rightsquigarrow [\mathbf{N^{h+1}}, \mathbf{L^{h+1}}].$$

Thus, by the transitivity of transitions, we see that

$$[\mathbf{N^0}, \mathbf{L^0}] \rightsquigarrow [\mathbf{N^{hmax}}, \mathbf{L^{hmax}}].$$

From the Reliable non-FIFO start lemma (Lemma 10.5), we get that,

$$[\mathbf{N_0}, \mathbf{G_N}\ \mathbf{G_L}] = [\mathbf{N^0}, \mathbf{L^0}].$$

From the Reliable non-FIFO last lemma (Lemma 10.6), we get,

$$[\mathbf{N}, \mathbf{L}] = [\mathbf{N^{hmax}}, \mathbf{L^{hmax}}].$$

The lemma follows from the last three relations. □

We are now ready to prove our main theorem for reliable non-FIFO links, which states that we can drive the system to any possible acyclic global state. The proof uses the earlier concatenation construction to load the links with a node

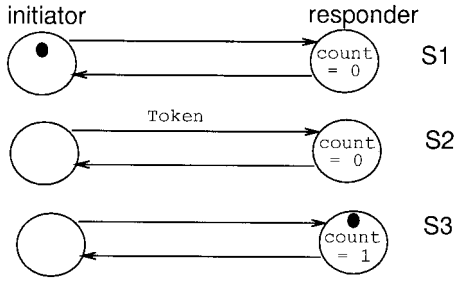

FIG. 20. Three possible consecutive states in a simple token passing protocol. $S1$ is an initial state. In state $S2$, the initiator passes the token to the link; in state $S3$, the responder receives the token and increments its counter.

and link state generator for the goal state; it then uses the previous lemma to play out these generators to result in the goal state.

THEOREM 10.9 (RELIABLE NON-FIFO ANY ACYCLIC STATE). *Let A be an arbitrary crashing protocol, and consider $A(RN)$ for a graph $G = (V, E)$. Consider (V_a, E_a), $V_a \subseteq V$, $E_a \subseteq E$ such that there is no cycle wholly among elements of $V_a \cup E_a$. Consider also an assignment of any possible node state s^i for every node $i \in V_a$ and of any possible link state $s^{i,j}$ for every link $(i, j) \in E_a$. Then there exists a state* $[\mathbf{N}, \mathbf{L}]$ *of $A(RN)$ such that $\mathbf{N}[i] = s^i$, $i \in V_a$, $\mathbf{L}[i, j] = s^{i,j}$, $(i, j) \in E_a$ and*

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

PROOF. Let $\mathbf{G_N}\ \mathbf{G_L}$ be the generator for the given acyclic state assignment where $\mathbf{G_N}\ \mathbf{G_L}$ is a concatenation of send sequences for $A(\mathrm{RN})$. Using the Reliable non-FIFO Concatenation Lemma (Lemma 10.3), we see that:

$$[\mathbf{N_0}, \mathbf{L_0}] \rightsquigarrow [\mathbf{N_0}, \mathbf{G_N}\ \mathbf{G_L}].$$

Using the Reliable non-FIFO playout lemma (Lemma 10.8), we see that

$$[\mathbf{N_0}, \mathbf{G_N}\ \mathbf{G_L}] \rightsquigarrow [\mathbf{N}, \mathbf{L}].$$

The theorem now follows from the last two transitions and the transitivity of transitions. □

11. *Proving that the Fault Span Hierarchy Is Strict*

Recall that, in Figure 1, we described the fault span of the different link models as a hierarchy, with the CAML model having the largest fault span and the CAM the least. We now show that the hierarchy is strict. We do so by first showing a protocol that cannot be driven to any possible state in the CAMO model; we then show a protocol which cannot be driven to any acyclic state in the CAM model.

11.1. COUNTEREXAMPLE FOR CAMO. Figure 20 shows a simple two node protocol that *cannot* be driven to any possible global state in the CAMO model. This shows that CAML is *strictly* more adversarial than CAMO.

Figure 20 shows three consecutive global states in a simple token passing protocol between two nodes, an initiator and a responder. On a crash, the initiator sets its token flag to be true (shown by a black dot). On a crash, the

```
There are two nodes i and r, each with a token bit token. r also has a counter count_r
token indicates whether a node has the token; count_r is the number of times r has
received the token since last crash.
Node i is the initiator of token passing after a crash;
Node r is the responder and waits for a token
In the initial state: token_i = true, token_r = false, count_r = 0, and the links are empty

crash_i (* Node i crashes, input action *)
   Effects: token_i = true;

crash_r (* Node r crashes, input action *)
   Effects: token_r = false; count_r = 0;

send_{i,r}(token) (* Node i sends a token to node r *)
   Pre: token_i = true;
   Effect: token_i = false;

send_{r,i}(token) (* Node r sends a token to node i *)
   Pre: token_r = true;
   Effect: token_r = false;

receive_{i,r}(token) (* Node r receives a token from node i *)
   Effects: token_r = true; count_r = count_r + 1;

receive_{r,i}(token) (* Node i receives a token from node r *)
   Effects: token_i = true;
```

FIG. 21. Counterexample Protocol for CAMO.

responder sets its token flag to be false and initializes a counter called *count* to 0. Each node can pass the token (by sending a *Token* packet on the link) when it has its token flag set; on doing so, it sets its *token* flag to *false*. The responder counter *count* is incremented every time the responder receives a *Token*; thus it keeps track of the number of tokens received since the last responder crash. Let us say there is a token in the system in some state if either node has its *token* flag true or there is a *Token* packet in either channel.

The code for the protocol is shown in Figure 21. The token flag at the initiator is $token_i$, and the flag and counter at the responder are $token_r$ and $count_r$. We define the token passing system to be the concatenation of the initiator and responder automata specified in Figure 21 together with two non-FIFO links $RN^{i,r}$ and $RN^{r,i}$.

Figure 20 only shows a simple execution. If the initiator crashes after sending a *Token*, we can clearly create more than one token; similarly, a responder crash can cause the loss of a token. We show, however, that there is at least one possible global state that is impossible to reach.

THEOREM 11.1. *There is a possible global state that cannot be reached by an execution of the token passing protocol of Figure* 21 *working in the CAMO model.*

PROOF. We show that any possible state s is impossible to reach as long as there is no token in state s and $count_r > 0$. It is easy to verify that s is a possible state: first, the node i and link states can be produced in state $S3$ of Figure 20; secondly, a node r state in which $count_r = c > 0$ will clearly occur in a crashless execution in which node r receives the token c times.

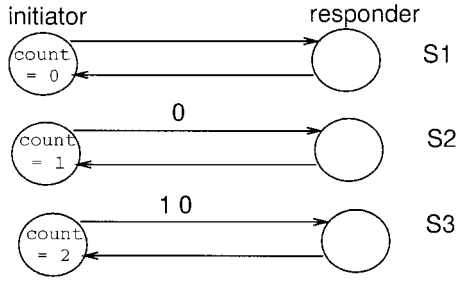

FIG. 22. Three possible consecutive states in a simple counter emitting protocol. $S1$ is an initial state; in $S2$, the initiator sends packet 0 and increments its counter; in $S3$, the initiator sends packet 1 and increments again.

We show that s is impossible to reach by describing an invariant that is true even in executions in which there are crashes. The invariant is: If $count_r > 0$, then there is at least one token in the system. If we can prove that this invariant holds for all reachable states of our token passing system, then state s is clearly not a reachable state because it violates the invariant.

We prove the invariant by induction of the length of an execution of the token passing system. For the base case, observe that the invariant holds trivially in the initial state as there is a token at i. For the inductive step, we show that if the invariant holds in state s_i, then it will hold for s_{i+1}, for any possible transition (s_i, a_i, s_{i+1}). We consider cases for action a_i.

It is easy to see that after any *send* action, there is a token in the link. Similarly after any *receive* action, there is a token at the node on the receiving end of the link. Thus the invariant holds trivially in s_{i+1} if a_i is a *send* or *receive*. If $a_i = crash_r$, the invariant hold trivially in s_{i+1} because $count_r = 0$ in s_{i+1}. If $a_i = crash_i$, the invariant hold trivially in s_{i+1} because $token_i = true$ in s_{i+1} which implies that there is a token in the system.

Thus, we have proved that the invariant is true in all reachable states of the token passing system. But s is a possible state that does not satisfy the invariant. Thus, there is at least one possible state that is not a reachable state of the token passing system. □

11.2. COUNTEREXAMPLE FOR CAM. We show a simple two-node protocol whose links *cannot* be driven to any possible state in the CAM model. This shows that CAMO is *strictly* more adversarial than CAM. Figure 22 shows three consecutive global states in a simple counter emitting protocol between two nodes, an initiator and a responder. On a crash, the initiator sets a counter $count_i = 0$. The initiator's only action is to send a packet containing its current counter and to then increment its counter. The initiator ignores all responder packets. The responder ignores all initiator packets and has no state variables.

The protocol code is shown in Figure 23. We define the counter emitting system to be the concatenation of the initiator and responder automata specified in Figure 23 together with two reliable FIFO links $R^{i,r}$ and $R^{r,i}$

Figure 22 shows a simple execution. The initiator can crash at any time and reset to 0 and send a 0 packet. Thus, one can easily create sequences of consecutive numbers on the links followed by a second such sequence starting with 0. However, we show that we cannot create arbitrary sequences of numbers.

```
There are two nodes i and r. i has a counter count_i.
count_i is the number of packets i has sent since its last crash.
In the initial state count_i = 0 and the links are empty

crash_i (* Node i crashes *)
   Effects: count_i = 0;

crash_r (* Node r crashes *)
   Effects: None (* Node r has no state variables! *)

send_{i,r}(c) (* Node i sends a counter to node r *)
   Pre: c = count_i;
   Effect: count_i = count_i + 1;

receive_{i,r}(x) (* Node r receives a packet from node i *)
   Effects: None
```

FIG. 23. Counterexample Protocol for the CAM model.

THEOREM 11.2. *There is a possible link state on the link from initiator to responder that cannot be reached by an execution of the counter emitting protocol working in the CAM model.*

PROOF. Our proof is again constructive. Consider a possible link state of link (i, r) (more precisely, we mean a state of $R^{i,r}$) which contains two consecutive packets (recall that the link is FIFO) that have numbers m and n, such that $m = n$ and $n > 0$. This is easily shown to be impossible to reach because all reachable states of the protocol satisfy the following invariant (even in executions with arbitrary crashes): If (i, r) contains two consecutive packets with numbers m and n, and such that m is closer to the responder than n, then either $n = m + 1$ or $n = 0$.

To prove this invariant (call it $I1$), we need a supporting invariant (call it $I2$) which states that if m is the counter of the last packet stored in link (i, r), then $count_i$ is either equal to $m + 1$ or 0.

It is easy to see that both invariants hold in the initial state and are maintained by $crash_i$ actions. Finally, any $send_i$ action in state s_i will preserve $I1$ and $I2$ in s_{i+1}, because $I2$ holds in s_i. Other actions do not affect the invariant. □

12. *Designing Correct Protocols that Are Resilient to Crashes*

Our results indicate that the combination of asynchrony, crash failures, lossy links, and no NVM is particularly deadly: a sequence of crashes can drive a protocol into (essentially) arbitrary global states. An impossibility result does not, however, mean that it is impossible to build correct crash resilient protocols. Rather an impossibility result highlights what must be changed in order to build a correct protocol. The two standard ways to get around an impossibility result are to either *change the assumptions* or *relax the specification.*

For example, if we change the assumptions behind our result and assume nonvolatile memory after a crash, then it is well known that protocols can be made resilient to node crashes. For instance, Baratz and Segall [1988] showed that it is possible to build a reliable Data Link protocol with a single bit of NVM at each node. Later, Finn [1979] and Awerbuch et al. [1987] showed how to make any network protocol resilient to crash failures using a reliable Data Link protocol. Thus, given the cheapness and availability of small amounts of NVM,

our results indicate that NVM is a cheap form of insurance for protocols that wish to be crash-resilient.

Assuming bounded message lifetimes is another way to change the assumptions. For example, Attiya et al. [1995] introduce a grace period between crashes. They also introduce a third way to change the assumptions by assuming that links have bounded capacity. Recall that our constructions depend crucially on the ability to build up potentially unbounded (though finite) sequences of messages on each link.

A different way to avoid the consequences of our impossibility result is to relax the correctness specification to only require *eventual* or *probabilistic* correctness. For example, we can design protocols to be self-stabilizing, in which case they eventually recover from an arbitrary state (including any state that can occur after a sequence of crashes) to a good state. Afek and Brown [1993] showed how to make a self-stabilizing data link and token ring protocol over an unreliable communication medium. However, self-stabilizing protocols have to tolerate some bad behavior before eventually converging to a good state. Thus, the existence of self-stabilizing Data Protocols does not contradict our result. Similarly, the existence of randomized protocols (that work correctly, with high probability, even in the face of crash failures) does not contradict our result.

13. *Conclusions*

A common line of research into fault-tolerance is to either design protocols that are resilient to faults or to show impossibility results for a particular set of faults. Our approach is to find the fault span, or the set of reachable states, for a particular fault model. The approach is similar to string generation problems in complexity theory. We want to see what strings (i.e., global states) we can generate, given a set of string manipulation operations (i.e., faults and normal protocol actions such as sending and receiving packets) and a set of restrictions on sequences of operations (e.g., restrictions imposed by the model based on synchrony and partial synchrony). The asynchronous model is best suited for this approach because it imposes the least restrictions on the ordering of operations.

Fault-spans can provide insight into possible failure modes of protocols. For instance, we know that in the CAML model (Crashing, Asynchronous, Memoryless and Lossy), a protocol that cannot deal with being in any possible state, is guaranteed to be incorrect. This can be used to prove the celebrated FLMS result, which in turn shows that a widely used Data Link protocol (HDLC) is incorrect. As another example, we showed that token passing and resource allocation protocols will work incorrectly in this model.

Many real-life protocols have possible states that include states in which they deadlock. Rather than do extensive state-space exploration techniques (which work only for finite state models), we can show that these protocols will not work in the CAML model. As we described above, we can only obtain a correct protocol for the CAML model by changing one or more of the assumptions or by relaxing the specification. An example of the former is to require NVM; an example of the latter is to design self-stabilizing protocols which only guarantee eventual correctness.

Some readers may argue that the constructions used to show faulty behavior are extremely involved and are so are "unlikely" to occur in practice. But the

major difficulty with this argument is making the term "unlikely" precise. If it were possible to estimate probabilities of crashes and asynchronous behaviors, it may be possible to show that our faulty executions have low probability of occurrence. However, very little knowledge of such probability distributions exists. Further, we have made no attempt to show that our constructions are optimal. Perhaps there are shorter, "more likely" constructions that can drive the system to the same target state.

In the absence of a definite and small probability that can be assigned to such erroneous executions, the only practical alternative is to conclude that the protocol does not work. Given such a lurking danger and the high cost of network failures, we believe that most practitioners (after such a result is pointed out) would rather switch to a protocol that is certifiably correct, as long as the correct protocol is reasonably inexpensive. Fortunately, the literature shows that changing the system assumptions (i.e., adding NVM) does allow the design of correct protocols at little cost.

The CAML result also shows that self-stabilization is not just limited to "recovery from bizarre faults". Even simple and commonly occurring failure modes like node crashes can conspire to drive systems into arbitrary states. This connection between self-stabilization and crash fault-tolerance is one that seems worthy of further exploration.

From a theoretical standpoint, computing the set of reachable states is considered difficult because it does not lend itself to familiar inductive approaches. The fault span in the CAML model is very close to the maximum possible. One might wonder whether there are nontrivial characterizations of the set of reachable states for other models? Our results for other link models show that the fault span approach can apply to other models. The results for the CAMO model show that we can drive any acyclic subset of the nodes and links to arbitrary states; the results for the CAM model show that we can drive nodes to arbitrary states. We also provide counterexamples to show that we cannot control all nodes and all links in the CAM and CAMO models. These results, summarized in Figure 1, also provide a form of sensitivity analysis for the CAML result. They show that the ability of links to lose packets is crucial for the CAML result.

We have not studied fault-spans using other models of faults or asynchrony though we believe that such study could provide new insights. There are several other interesting combinations besides the CAML model that are appropriate for modeling networks. We can select among synchronous, asynchronous, and partially synchronous (e.g., for real-time systems) timing models. We can select various combinations of link faults (non-FIFO, crashes, corruption, insertion, loss, duplication) and node faults (crash, arbitrary memory failure, Byzantine faults).

In terms of our approach, we believe that a characterization in terms of reachable global states was more fruitful than a characterization using behaviors that was used in Fekete et al. [1993]. It also helped to pose a more general question than a more specific one. We initially asked ourselves if resets were impossible in this model, before we considered the easier (but more general) question of fault spans.

Our notation representing global states using node and link state *vectors* considerably improved the readability of our proofs. The vector notation allowed compact manipulation of global states. Also, after defining facets of the link model using a crash, locality and loss lemma, the remainder of the proof was

completely algebraic and model independent. We hope that this notation may be useful elsewhere.

In conclusion, we believe we now understand the possible effects of crash failures when no NVM is available. Our result provides useful insight for protocol designers, and a precise justification of earlier folk theorems. We hope it will lead to further investigation of fault spans in other models.

ACKNOWLEDGMENTS. We would like to thank Alan Fekete for his detailed comments and helpful suggestions. We also thank Yishay Mansour for his helpful comments.

REFERENCES

AFEK, Y., AWERBUCH, B., AND GAFNI, E. 1987. Applying static network protocols to dynamic networks. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science* (Oct.). IEEE Computer Society Press, Los Alamitos, Calif. pp. 358–370.

AFEK, Y., AND BROWN, G. M. 1993. Self-stabilization over unreliable communication media. *Distr. Comput. 7*, 1, 27–34.

ATTIYA, H., DOLEV, S., AND WELCH, J. L. 1995. Connection management without retaining information. *Inf. Comput. 123*, 2, (Dec.), 155–171.

BARATZ, A., AND SEGALL, A. 1988. Reliable link initialization procedures. *IEEE Trans. Commun.* (Feb.), 144–153.

DIGITAL EQUIPMENT CORPORATION. 1983. *Phase IV NSP Functional Specification*. Digital Order Number AA-X439A-TK.

FEKETE, A., LYNCH, N. A., MANSOUR, Y., AND SPINELLI, J. 1993. The impossibility of implementing reliable communication in the face of crashes. *J. ACM, 40*, 5 (Nov.).

FINN, S. C. 1979. Resynch procedures and a fail-safe network protocol. *IEEE Trans. Commun. COM-27*, 6 (June), 840–845.

JAYARAM, M. 1996. Fault span of crash failures. M.S. Thesis, Washington Univ. St. Louis, MO.

JAYARAM, M., AND VARGHESE, G. 1997. The complexity of crash failures. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (Santa Barbara, Calif., Aug. 21–24). ACM, New York, 179–188.

LYNCH, N. A., AND TUTTLE, M. R. 1989. An introduction to input/output automata. *CWI Quarterly 2*, 3, 219–246.

LYNCH, N. A. 1996. *Distributed Algorithms.* Morgan-Kaufman, San Francisco, Calif.

MCQUILLAN, J. M., RICHER, I., AND ROSEN, E. C. 1980. The new routing algorithm for the arpanet. *IEEE Trans. Commun. COM-28*, 5 (May), 711–719.

TANNENBAUM, A. 1996. *Computer Networks*, 3rd ed. Prentice-Hall, Upper Saddle River, N.J.

WATSON, R. W. 1981. Timer based mechanisms in reliable transport protocol connection management. *Comput. Netw. 5* (Feb.), 47–56.

RECEIVED JULY 1997; REVISED JULY 1999; ACCEPTED JULY 1999