Crash failures can drive Protocols to Arbitrary States

Mahesh Jayaram* George Varghese

Department of Computer Science
Washington University
St. Louis, MO 63130

Abstract

A *crashing* network protocol is an asynchronous protocol that has no non-volatile memory at nodes that can survive a node crash and restart. Thus after a crash and restart, a node in such a protocol returns to a prespecified start state. We consider crashing protocols that work with links that can drop packets. Our main theorem states that such crashing protocols can be driven by a sequence of crashes to any global state, where each node is in a state reached in some (possibly different) run, and each link has an arbitrary mixture of packets sent in (possibly different) runs.

Our theorem can be used to give an alternate proof of an earlier result, due to Fekete et al, which states that there is no correct crashing Data Link Protocol. Our theorem can also be used to derive *new* results. We prove that there is no correct crashing token passing protocol. We also prove that there is no correct crashing protocol for many other resource allocation protocols such as k-exclusion, and the drinking and dining philosophers problems. Our theorem shows that existing crashing network protocols (that are widely deployed) are either incorrect or are self-stabilizing.

Supported by NSF Grant NCR-9405444

PODC'96, Philadelphia PA, USA
© 1996 ACM 0-89791-800-2/96/05..$3.50

1 Introduction

We consider asynchronous network protocols that work with faulty components: links that can lose packets, and nodes that can crash and restart. Many common network protocols (e.g., HDLC, IP, the OSI and DECNET Routing protocols [Tan81]) come under this category.[1] Crash failures, where a node crashes in the middle of a protocol, are a common cause of protocol failures. Frequently, protocols that seem to work for years are driven into deadlocked states by a series of crashes.

Many of these protocol specifications do not require the nodes to have non-volatile memory. Thus if a node crashes and restarts, it can lose all memory of its previous state. After a 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 non-volatile memory (e.g., disk) is cheap and provides protection against crash failures.

However, many early protocol implementations were on stand-alone devices (e.g., routers) that did not have a disk. Adding a disk was precluded by the expense, and sometimes by the physical configuration of the device. Thus many network protocols like IP and HDLC *do not require* that nodes have non-volatile memory (NVM). Recently, cheaper electronic forms of NVM (e.g., NVRAM) have become available. Nevertheless, results about protocols that do not require NVM are interesting, because many existing protocols are in this category.

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

Baratz and Segall [BS88] showed that the widely deployed Data Link protocol HDLC could work incorrectly if nodes did not keep NVM. Later, Fekete et al[FLMS93] showed that no Data Link protocol could work correctly under these assumptions. Attiya et al[ADW95] have proved related impossibility results for Transport protocols. In this paper, we investigate the power of crash failures for protocols other than just Data Link or Transport protocols. Our main theorem can be used to give an alternate proof of the Data Link result in [FLMS93]. It can also be used to show new results: for instance, the impossibility of token passing or resource allocation with crash failures and no NVM.

Our results indicate that the combination of asynchrony, unlimited link storage, crash failures and no NVM is particularly deadly: a sequence of crashes can drive a protocol into (essentially) arbitrary global states. On the other hand, it is well-known that protocols can be made resilient to node crashes with the use of NVM. Baratz and Segall[BS88] describe a reliable Data Link protocol that uses a single bit of NVM at each node. Other papers ([Fin79, AAG87] show how to make any network protocol resilient to crash failures using a reliable Data Link protocol. Given the cheapness and availability of small amounts of NVRAM, our results indicate that NVRAM is a cheap form of insurance for protocols.

Protocols can also deal with arbitrary states by being self-stabilizing, so they can recover from an arbitrary state. Afek and Brown [AB93] describe self-stabilizing Data Link and Token Passing protocols that work over an unreliable link. However, self-stabilizing protocols are allowed to exhibit bad behavior before converging to a good state. These results do not contradict our main theorem because our correctness criteria disallows incorrect behavior at any time.

Our paper 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 deal with all 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 suggest that investigating fault-spans for other faults and other models is a useful direction.

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 and some useful notation in Section 3 and culminates in the statement of the main theorem in Section 4. We describe applications of our theorem in Section 5, describe extensions to other models in Section 6 and state our conclusions in Section 7.

2 Intuition behind main theorem

In what follows, we do not distinguish between crashes and restarts. What we call a crash can be imagined to be a crash that is immediately followed by a restart. Intuitively, a *crashing protocol* is a protocol in which the nodes have no non-volatile memory (NVM); thus after a crash event, a node goes to a prespecified initial state.[2] Formal definitions are given later in the context of the I/O Automaton model. In this section, we describe the intuition behind the main result. We start by describing an intuitive way to understand the Data Link impossibility result of Fekete et al. We then show how to generalize the underlying construction to drive a protocol into (what we call) any *possible state.*

2.1 Data Link Impossibility

A Data Link protocol guarantees the transmission of a sequence of data items between a sender node and a receiver node without loss, misordering, or duplication. The links between the nodes may lose data items. If nodes do crash, a reasonable correctness condition is that all data items sent *after the last crash* (assuming crashes stop) are received successfully. The Data Link impossibility result [FLMS93] of Fekete et al (that we will call the FLMS result) shows that there is no correct crashing Data Link (DL) protocol.

In a typical DL protocol like HDLC, after a sender crash the sender sends a reset or handshake packet and waits for a handshake response from the receiver. The

[2] 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.

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.

2.2 Intuitive Statement of Result

We use CAML to denote an *A*synchronous model in which *C*rashes can occur, nodes are *M*emoryless after crashes, and packets can be *L*ost on links. The FLMS result shows that reliable Data Links are impossible in the CAML model. Are there other protocols that do not work in the CAML model? There are, but to state our result we need some definitions.

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

Possible states are not more restrictive than truly arbitrary global states. We can always modify a protocol to get rid of unreachable node states (since they do not occur normally anyway) and to ignore invalid packets (since they should not be received normally anyway). Thus there appears to be no loss of generality. The added checks are common-sense checks that should be added for fault-tolerance anyway.

Note that the 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 1. The top two global states are states that occur in (possibly different) finite executions. Given these two states, we can construct a possible global state by "mixing and matching". We pick one node state from one global state, and the other node state from the second global state; finally the links contain an arbitrary sequence of packets drawn from the packets on the links in the two global states.

Our main theorem essentially states: *Any crashing protocol that works in the CAML model can be driven into any possible protocol state.* Thus the fault-span of the CAML model is very large. The CAML model

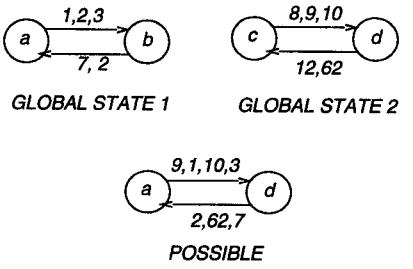

Figure 1: Two global states drawn from finite executions of the protocol and a possible state constructed by "mixing and matching" from the two global states.

is equivalent to arbitrary memory corruption at nodes and arbitrary packets on links.

2.3 Construction underlying new result

We describe the intuition for a two node protocol only. We begin with a construction that underlies the FLMS result. Consider a two-node protocol with a pair of unidirectional links. 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 2 shows an example execution of a 2 node protocol and a send sequence on the receiver-sender link. Notice that this sequence contains *all* packets sent on link L from the start of the execution.

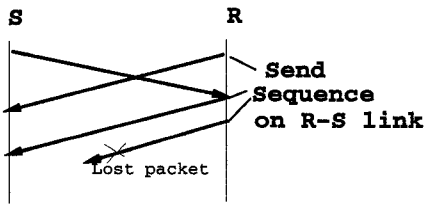

Figure 2: An example of a send sequence on a link.

The FLMS result constructs a series of crashes that leave the two node protocol in a state where all nodes are in initial states, and all links are empty except for a single link that has the entire sequence sent in a particular execution (Figure 3).

The essence of the FLMS construction is shown in Figure 4. A series of alternating crashes force a node R to send the first i packets of the sequence of normal crashless packets. This causes the other node S (after a crash) to send the packets that are needed that will force node R to send the first $i+1$ packets before crashing again. By continuing inductively, the re-

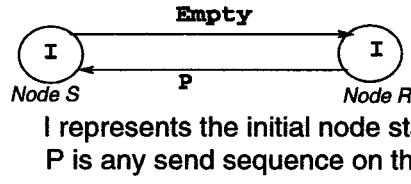

Figure 3: What the FLMS construction produces. The I at nodes represents the initial state (state that the node reverts to after a crash) of the corresponding node. P is a send sequence for the link.

ceiver is forced to emit the entire sequence of packets it would have emitted in a crash-free execution. Finally, we crash the sender and receiver. The result? The link from the receiver to sender has the complete sequence of packets sent in a crash-free 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 if the sender's first data item is lost, the sender will be fooled into thinking all is well.)

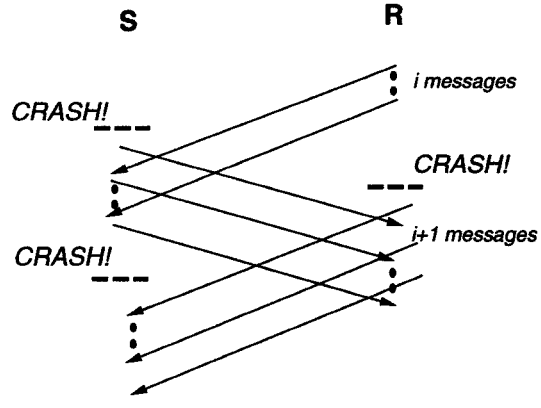

Figure 4: The essence of the FLMS construction: initializing a link with a sequence of old packets from a past incarnation.

We generalize the FLMS construction in order to initialize a link with a *concatenation* of send sequences as shown in Figure 5.

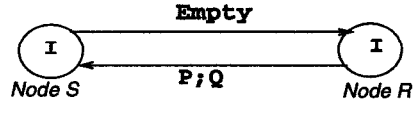

Figure 5: Our main construction is one that produces a concatenation of two send sequences.

For two nodes, we construct the concatenation, say $P;Q$, 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 the sender to send a sequence that produces the first $i+1$ packets of P, and so on. At the same time as we produce the first i packets of P, we crash the receiver and then produce the first i packets of Q. Thus two independent FLMS constructions are dovetailed (Figure 6).

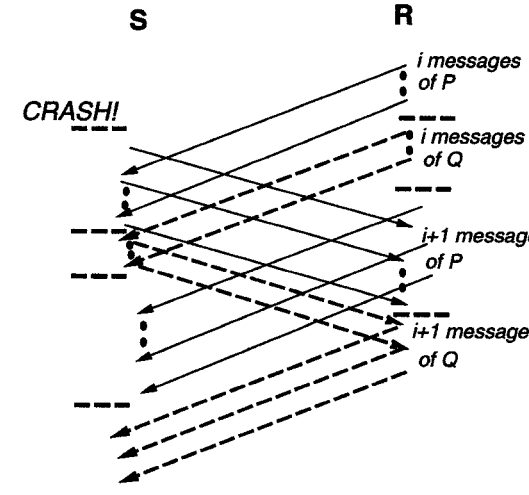

Figure 6: How concatenation works. The horizontal dashed lines represent crashes. The construction dovetails two independent FLMS constructions.

There are subtleties to the construction. First, if the sequences have different lengths, at some point we stop extending one sequence and keep increasing the longer sequence. Second, the construction does not generalize easily to multiple nodes and different topologies. In our formal development, we use a different construction (that essentially constructs the send sequence for *all* links of an execution at the same time) in place of the basic FLMS and concatenation constructions.

Using repeated concatenation, we can produce an arbitrary sequence of possible packets. By definition, any possible packet occurs in some send packet sequence. Thus we first concatenate the required number of send packet sequences (that contain the desired sequence as a subsequence). Then we lose the unnecessary packets to get the desired sequence.

Once we construct an arbitrary sequence of possible packets on a link, we can drive the 2-node system into any *possible* state. This is illustrated in Figure 7. 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 the second subgoal of finding a sequence of packets M (on the reverse link this time) that can cause the sender to emit the sequence L followed by the sequence Y we need for the goal state.

Generating L is easy because L is a subsequence of a send sequence; thus there must be some sequence of packets which 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 can get C, and finally obtain Y by losing packets. Similarly, we have a third subgoal (similar to the first subgoal) that shows that we can drive node S to state s with a sequence N.

We finish (see last frame in Figure 7) the construction by "loading" the reverse link with the sequence $M;N;X$ (where M is at the head of the link). We then first allow node S to receive M and emit $Y;L$ 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.

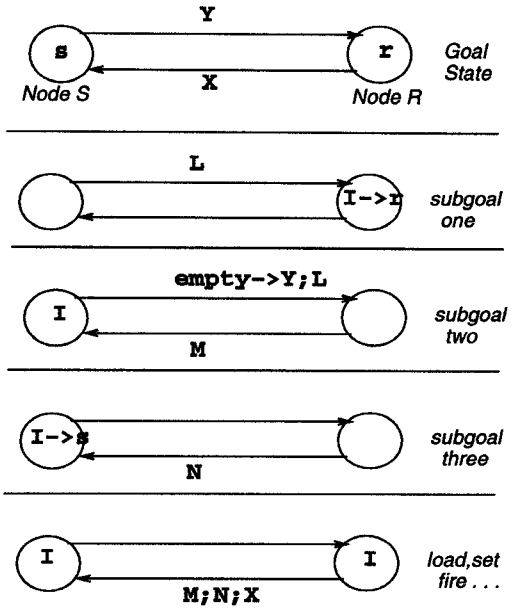

Figure 7: Driving a system into any possible state by loading one link with the appropriate sequence of packets and by playing out these packets.

3 Model

We assume familiarity with the Input-Output Automata (IOA) model [LT89] except for some basic notation. We model nodes and links by IOA.

Input-Output Automata (IOA): An IOA is defined by a *state set* S, an *action set* A, an *action signature* Z (that classifies the action set into input, output, and internal actions), a *transition relation* $R \subseteq S \times A \times S$, 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.[3] An *execution* is an execution fragment that begins with a start state.

There is a notion of *composition* [LT89] 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 with all states removed, and is denoted by $sched(\alpha)$. We say β is a *schedule* of A if β is the schedule of an execution of A. We say β is a *behavior* of A if β is the behavior of an execution of A. We define *fair schedule* and *fair behavior* analogously.

Modeling Crashing Network Protocols: We model the network system as a composition of automata, one

[3] The IOA model actually specifies fairness in terms of equivalence classes and the definition really applies to all actions in a class.

for each node and one for each unidirectional link between neighoring nodes. We model the network topology using a directed graph $G = (V, E)$. Nodes 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 $crash^i$.

We state our formal results only for FIFO unreliable links. Later, we informally describe our results for reliable links. Clearly impossibility results for FIFO links apply to non-FIFO links. Any reasonable unreliable FIFO link can be expected to satisfy the following properties: only packets which are sent are received, the link is FIFO, and the link is live — i.e., if a packet is sent an infinite number of times, it is received an infinite number of times. Let us call any sequence of *send* and *receive* actions which satisfies the above properties $\mathrm{U} - consistent$. Let us call a link automaton *universal* if its fair behaviors are all the sequences of actions which are $\mathrm{U} - consistent$. [FLMS93] describes a universal link automaton which we call $\mathrm{U} - universal^{u,v}$. We now describe our universal link automaton, $\mathrm{U}^{u,v}$ which models a link between nodes u and v. We have proved that $\mathrm{U}^{u,v}$ is also universal by proving that it is equivalent to $\mathrm{U} - universal^{u,v}$.

$\mathrm{U}^{i,j}$ has an input action $send^{i,j}(p)$ by which node i sends a packet to node j. It has a $receive^{i,j}(p)$ by which node j receives packet p. It also has an internal $lose$ action for losing packets. The state of $\mathrm{U}^{i,j}$ consists of a *queue* each element of which is a pair (p, k) where p is a packet and k is an integer, an array *count* of integers indexed by packet values, and an array *keep* 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* is empty and each entry $count[p]$ is zero. Thus each initial state is determined by *keep*. The only function of the packet counts and the *keep* array is to guarantee the liveness condition. The actions of $\mathrm{U}^{i,j}$ are as follows:

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

Our model includes a notion of fair execution, whose definition is not important to us; all we use is the fact that every finite execution has a fair extension. Our results concern reachable states of a system consisting of all nodes and links. 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 $(i, j) \in E$.

Vector Notation for Global States: We use vector notation to succintly 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 that has a component for each link. Fix a protocol automaton $A(\mathrm{U})$ for graph $G = (V, E)$. Formally, if $[\mathbf{N}, \mathbf{L}]$ denotes the state s of $A(\mathrm{U})$ then for all $i \in V, \mathbf{N}[i] = s \mid A^i$ and for all $(i, j) \in E$, $\mathbf{L}[i, j] =$ the sequence consisting only of packets in $s \mid \mathrm{U}^{i,j}$ i.e. if $s \mid \mathrm{U}^{i,j} = (p_1, k_1)(p_2, k_2)\ldots$ then $\mathbf{L}[i, j] = p_1 p_2 \ldots$. All vectors are written in bold.

$\mathbf{K} = \mathbf{L}\ \mathbf{M}$ is the *concatenation of link state vectors* $\mathbf{L}$ and $\mathbf{M}$ if $\mathbf{K}[i, j] = \mathbf{L}[i, j]\ \mathbf{M}[i, j]$, for all $i, j \in E$. Let $\mathbf{N_0}$ be the node state vector such that for every $i \in V$, $\mathbf{N_0}[i] =$ 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. The start

state of $A(\mathrm{U})$ for every execution is $[\mathbf{N_0}, \mathbf{L_0}]$. Given an execution α of $A(\mathrm{U})$ we use $acts(\alpha)$ to denote the sequence of actions in α. We use $acts^i(\alpha)$ to denote $acts(\alpha) \mid A^i$ (the subsequence of $acts(\alpha)$ projected on to A_i).

We describe state transitions of $A(\mathrm{U})$ 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 $A(\mathrm{U})$ from state $[\mathbf{N}, \mathbf{L}]$ to state $[\mathbf{N}', \mathbf{L}']$. The notation denotes that if there exists a finite schedule β which takes $A(\mathrm{U})$ to state $[\mathbf{N}, \mathbf{L}]$, then there exists a finite schedule $\beta\beta'$ which takes $A(\mathrm{U})$ 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 which takes $A(\mathrm{U})$ 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 $[\mathbf{N}, \mathbf{L}] \rightsquigarrow [\mathbf{N}'', \mathbf{L}'']$.

To give the reader some feeling of the notation, we describe two basic lemmas that express the effect of a node crash at any node i and the ability of a link (i, j) to lose packets. The proofs follow directly from the definitions.

Lemma 3.1 (Crash):
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.

Lemma 3.2 (Loss):
Let $\mathbf{L}$ *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* $\mathbf{N}$, $[\mathbf{N}, \mathbf{L}] \overset{\beta}{\rightsquigarrow} [\mathbf{N}, \mathbf{L}']$, *where* $\mathbf{L}' = \mathbf{L}$ *except that* $\mathbf{L}'[i, j] = Q'$ *and* β *consists only of lose actions.*

4 Main Theorem

We state our result using IOA terminology. For any given finite execution $\alpha = s_0\ a_1\ s_1\ a_2\ s_2\ \ldots\ a_n\ s_n$ of $A(\mathrm{U})$ let $last\_state^j(\alpha)$ denote $s_n \mid A^j$, 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 $snd^{j,k}(\alpha)$.

A *possible node state vector* is a node state vector in which each node component is a final state node component of (possibly) different executions. More formally, a possible node state vector $\mathbf{N}$ of $A(\mathrm{U})$ is a node state vector such that for all $i \in V$, there exists some finite execution α^i of $A(\mathrm{U})$ such that $\mathbf{N}[i] = last\_state^i(\alpha^i)$ for some finite execution α^i of $A(\mathrm{U})$.

A *possible packet* on a link is a packet which is sent on that link in some execution. 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 sequence* on link $\mathrm{U}^{i,j}$ of $A(\mathrm{U})$ is defined as a sequence of possible packets for that link.

A *possible link state vector* $\mathbf{L}$ of $A(\mathrm{U})$ is a link state vector s.t. for all $(i, j) \in E$, $\mathbf{L}[i, j]$ is a possible sequence on link $\mathrm{U}^{i,j}$ of $A(\mathrm{U})$. Finally we define a *possible state* of $A(\mathrm{U})$ as a state $[\mathbf{N}, \mathbf{L}]$ s.t. $\mathbf{N}$ is any possible node state vector and $\mathbf{L}$ is any possible link state vector.

We call our main theorem the Any State Theorem because it expresses the ability to drive a crashing protocol to any possible state. Expressed in our notation, we have,

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

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

We omit a proof of this Theorem for lack of space. The proof is simple and intuitive, and carried out almost entirely using the vector notation. A detailed proof can be found in [JV96].

5 Applications

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 specified using executions) and a Data Link (correctness specified using behaviors). We show how the token passing proof can be extended to show that there is no crashing solution to the Dining or Drinking Philosophers problem.

5.1 Token Passing Protocols

We now prove that it is impossible to construct a reliable crashing token passing protocol. We first define

a token passing protocol and then state its correctness criteria.

We define a token passing protocol for a graph[4] $G = (V, E)$ as a crashing protocol (see Section 3) $T = (A^1, A^2, \ldots, A^n)$ where $n = \mid V \mid$ 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(U) be a crashing automaton for T for graph $G = (V, E)$.

T(U) is said to be correct if it satisfies two properties: **T1) Safety:** For all executions α of T(U) and any state s which occurs in α, at most one of the predicates $token^i(s \mid A^i) = true$. **T2) Liveness :** In any fair execution α of T(U), for all $i \in V$, there exists infinitely many states s such that $token^i(s \mid A^i) = true$.

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 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).

Theorem 5.1 (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(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 $token^i(s_i \mid A^i) = true$ and $token^j(s_j \mid A^j) = true$ for $i \neq j$, $i, j \in V$. Consider the state s such that $s \mid A^i = s_i \mid A^i$ and $s \mid A^j = s_j \mid A^j$ and $s \mid A^k = s_0 \mid A^k$ for $k \neq i, j$ and $k \in V$, and where s_0 is the initial state of T(U). Clearly s is a possible state of T(U). Applying the Any State Theorem, there exists a finite execution α' which takes T(U) from state s_0 to state s, such that s violates the safety property T1. Thus T is incorrect. ∎

5.2 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 [Lyn96]) 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, the processes that share a resource are connected by an edge in the topology graph, and thus sets containing a node and its neighbors are exclusion sets. We can 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 $E \subseteq V$. 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 to a state which violates its safety property. We omit details.

5.3 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 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(U) be the crashing automaton for D and $G = (V, E)$. We only present an informal rendering of the proof here.

Without crash actions we would require that every message sent is received. The specification with crash actions is more delicate (see [FLMS93]) but we only need two reasonable correctness conditions. A data link automaton is correct if for all behaviors: **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 — i.e., 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.

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. We claim

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

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.)

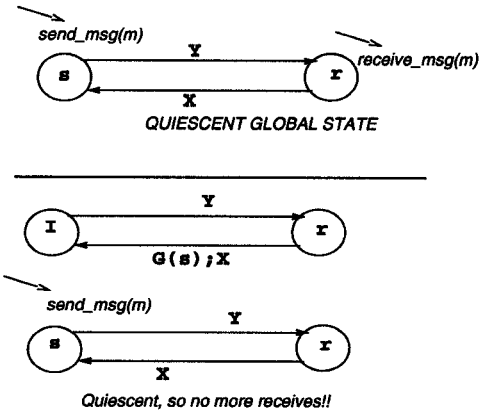

Figure 8: Data Link Impossibility Proof consists of finding a behavior in which a message is sent and then the system is driven to a quiescent state in which the message is not delivered.

The impossibility proof is illustrated in Figure 8. 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 $s'|U^{r,t} = G(a)\ X$ as shown in the figure. We 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 for $G(a)$ 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.

6 Generalizations

We discuss generalizations of the main result (Any State Theorem) that can apply to other models (besides IOA) and other fault models (besides the CAML model)

The Any State Theorem appears to apply to any state machine model of a network protocol that satisfies the Crash Lemma (Lemma 3.1), the Loss Lemma (Lemma 3.2), and a locality lemma. These lemmas correspond to facets of the CAML model. The Crash Lemma expresses the Crash and Memoryless properties, the Loss Lemma represents packet loss, and the locality lemma reflects both the locality and asynchrony of the model. These lemmas hold in the IOA model of a crashing protocol.

We can generalize beyond the CAML model. For example, we can show that if use a CAM model (i,e., same as CAML except no packet loss allowed), then the fault span is dramatically reduced. We find that in the CAM model we can drive a pair of nodes into arbitrary combinations of valid states but *we cannot control the links as well.* This suffices to show that there is no crashing token passing or resource allocation protocol even in the CAM model. We can also show examples of protocols that cannot be driven into possible states in the CAM model. This shows that the CAML model is strictly more powerful than the CAM model. We have not studied fault-spans using other models of faults or asynchrony though we believe that such study could provide new insights.

7 Conclusions

It is common to design fault-tolerant protocols or to show impossibility results for a particular fault model. Our approach is to find the fault span for a given fault model, and is similar to string generation problems in complexity theory. We determine the set of strings (i.e., global states) we can drive a system into, given a set of string manipulation operations (e.g., faults, sending packets) and a set of restrictions (e.g., restrictions imposed by synchrony). The asynchronous model is easiest to work with because it imposes the least restrictions.

Fault-spans provide insight into failure modes of protocols. Thus in the CAML model, a protocol that cannot deal with being in any possible state is incorrect. This can be used to prove the celebrated FLMS result, which shows that a widely used Data Link protocol (HDLC) is incorrect. As another example, we

showed that reliable token passing and resource allocation protocols are impossible in this model.

Many real-world protocols include deadlock states among their possible states. Without using state-space exploration (which works only for finite state models), we can show that these protocols will not work in the CAML model. Our result implies that: *A correct protocol for the CAML model must be self-stabilizing — i.e., it must be prepared to recover from every possible state.* If we do not want to design self-stabilizing protocols, *we must remove one or more of the assumptions of the CAML model.*

One way to design correct protocols is to remove the memoryless assumption. For instance, it is well known that we can design correct Data Links or even Transport Protocols using non-volatile memory. A second way is to remove the asynchrony requirement and assume a bound on packet delay T. Then the protocol can wait T time after a crash to allow old packets to die out. Interestingly, similar results hold for token passing. Most real token passing protocols assume a maximum packet delay bound and send a RESET signal around the ring and wait the time bound. However, our results indicate that another and faster way is to use an incarnation number that is stored in NVRAM. Finn [Fin79] showed how to modify any protocol to work with node crashes, assuming either NVRAM or a crash-resilient Data Link. Finn's result uses a new incarnation number after any crash. This result is consistent with our result because it requires a violation of the CAML model.

The CAML result also shows that self-stabilization is not 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 seems worth exploring.

Acknowledgements : We would like to thank Alan Fekete and Pete McCann for their detailed comments and helpful suggestions. We also thank Yishay Mansour for his helpful comments.

References

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.

[BS88] A. Baratz and A. Segall. Reliable link initialization procedures. *IEEE Trans. on Commun.*, February 1988.

[Fin79] Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.

[FLMS93] Alan Fekete, Nancy A. Lynch, Yishay Mansour, and John Spinelli. The Impossibility of Implementing Reliable Communication in the Face of Crashes. In *JACM*, vol 40, No. 5, November 1993. Also, Technical Memo MIT/LCS/TM-355, May 1988.

[JV96] Mahesh Jayaram and George Varghese. Crash failures can drive Protocols to Arbitrary States (Detailed Version) Available at ftp::://dworkin.wustl.edu/dist/george/crash.ps.Z.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Lyn96] Nancy A. Lynch Distributed Algorithms Forthcoming textbook published by *Morgan Kaufman*

[Tan81] A. Tannenbaum. *Computer Networks.* Prentice Hall, 1981.

[AB93] Y. Afek, G.M. Brown. Self-stabilization over unreliable communication media. In *Distributed Computing*, vol 7, No. 1, 1993.

[ADW95] Hagit Attiya, Shlomi Dolev, and Jennifer L. Welch. Connection Management Without Retaining Information. In *Information and Computation*, vol 123, No. 2, pp. 155-171, Dec. 1995.