Lecturer: George Varghese

Lecture 23: Self-Stabilization

Designing network protocols is complicated by three issues: parallelism, asynchrony, and fault-tolerance. This course has already covered techniques for making protocols robust against Byzantine faults. The Byzantine fault model is attractive because it is general – the model allows a faulty process to *continue* to behave arbitrarily. In this lecture, we will study another general fault model – the *self-stabilization* model – which is attractive from both a theoretical and practical viewpoint.

We will compare the two models a little later but for now let's just say that self-stabilization allows an *arbitrary* number of faults that *stop* while Byzantine models allow a *limited* number of faults that *continue*. Stabilizing solutions are also typically much cheaper and have found their way into real networks. In this lecture we will describe self-stabilization using two network models – an elegant shared memory model and a more practical network model. We will also describe three general techniques for making protocols stabilizing – *local checking and correction, counter flushing,* and *timer flushing*.

1 Introduction

1.1 Door Closing and Domain Restriction

Today we will focus on the ability of network protocols to stabilize to "correct behavior" after *arbitrary* initial perturbation. This property was called *self-stabilization* by Dijkstra [Dij74]. The "self" emphasizes the ability of the system to stabilize by *itself* without manual intervention.

A story illustrates the basic idea. Imagine that you live in a house in Alaska in the middle of winter. You establish the following protocol (set of rules) for people who enter and leave your house. Anybody who leaves or enters the house must shut the door after them. If the door is initially shut, and nobody makes a mistake, then the door will eventually return to the closed position. Suppose, however, that the door is initially open or that somebody forgets to shut the door after they leave. Then the door will stay open until somebody passes through again. This can be a problem if heating bills are expensive and if several hours can go by before another person goes through the door. It is often a good idea to make the door closing protocol self-stabilizing. This can be done by adding a spring (or automatic door closer) that constantly restores the door to the closed position.

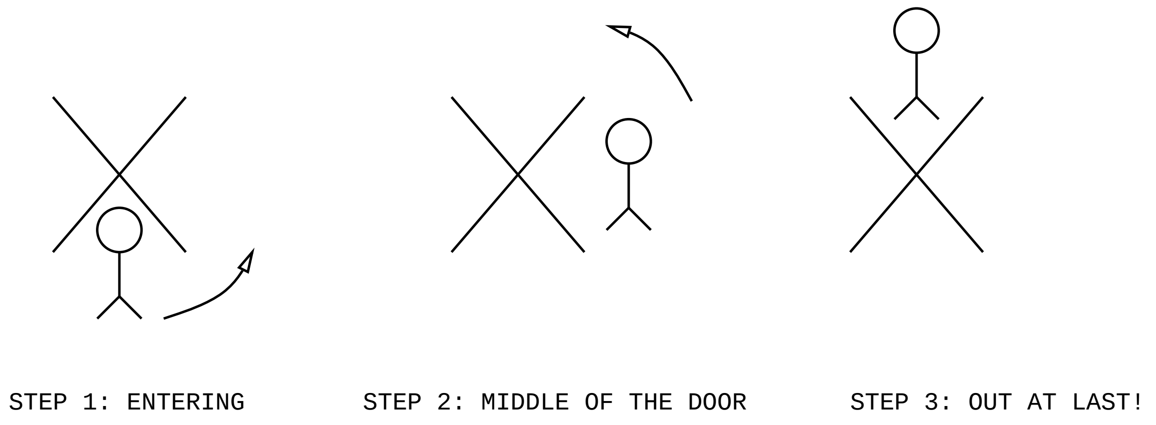

Figure 1: Exiting through a revolving door.

In some cases, it is possible to trivialize the problem by hardwiring relationships between variables to avoid illegal states – such a technique is actually used in a revolving door! A person enters the revolving door Figure 1, gets into the middle of the door, and finally leaves. It is physically impossible to leave the door open and yet there is a way to exit through the door.

This technique, which we will call *domain restriction*, is often a simple but powerful method for removing illegal states in computer systems that contain a single shared memory. Consider two processes A and B that have access to a common memory as shown in the first part of Figure 2. Suppose we want to implement mutual exclusion by passing a token between A and B.

One way to implement this is to use two boolean variables $token_A$ and $token_B$. To pass the token, Process A sets $token_A$ to *false* and sets $token_B$ to *true*. In a self-stabilizing setting, however, this is not a good implementation. For instance, the system will deadlock if $token_A = token_B = false$ in the initial state. The problem is that we have some extra and useless states. The natural solution is to restrict the domain to a single bit called *turn*, such that $turn = 1$ when A has the token and $turn = 0$ when B has the token. By using domain restriction, [1] we ensure that *any possible state is also a legal state.*

It is often feasible to use domain restriction to avoid illegal states *within a single node* of a computer network. Domain restriction can be implemented in many ways. The most natural way is by restricting the number of bits allocated to a set of variables so that every possible value assigned to the bits corresponds to a legal assignment of values to each of the variables in the set. Another possibility is to modify the code that reads variables so that only values within the specified domain are read.

[1] In this example, we are really *changing* the domain. However, we prefer the term domain restriction.

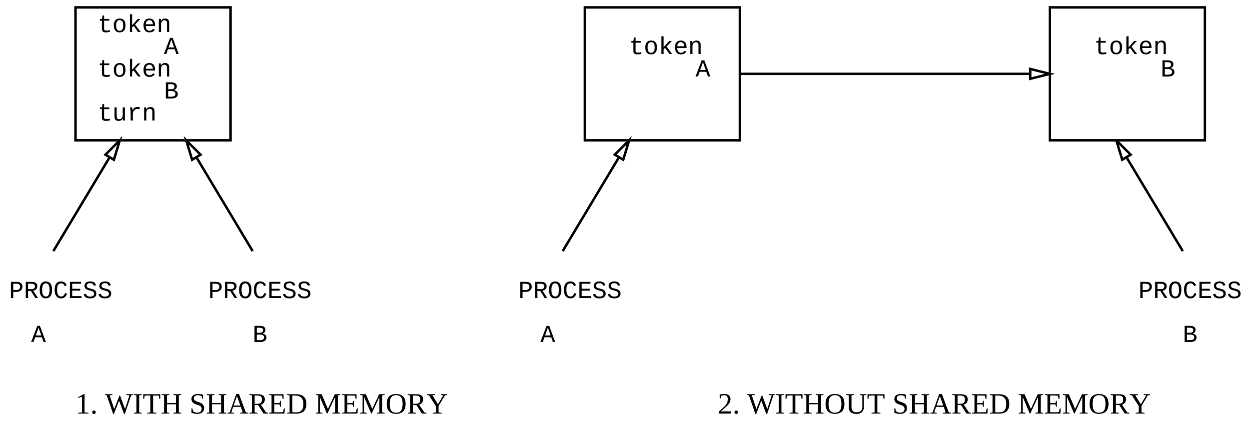

Figure 2: Token passing among two processes

Unfortunately, domain restriction cannot solve all problems. Consider the same two processes A and B that wish to achieve mutual exclusion. This time, however, (see Figure 2, Part 2) A and B are at two different nodes of a computer network. The only way they can communicate is by sending *token* messages to each other. Thus we cannot use a single *turn* variable that can be read by both processes. In fact, A must have at least two states: a state in which A has the token, and a state in which A does not have the token. B must also have two such states. Thus we need at least four combined states, of which two are illegal.

Thus domain restriction at each node *cannot prevent illegal combinations across nodes*. We need other techniques to detect and correct illegal states of a network. It should be no surprise that the title of Dijkstra's pioneering paper on self-stabilization [Dij74] was "Self-Stabilization in spite of Distributed Control."

1.2 Self-Stabilization is attractive for Networks

We will explore self-stabilization properties for computer networks and network protocols. A computer network consists of nodes that are interconnected by communication channels. The network topology (see Figure 3) is described by a graph. The vertices of the graph represent the nodes and the edges represent the channels. Nodes communicate with their neighbors by sending messages along channels. Many real networks such as the ARPANET,

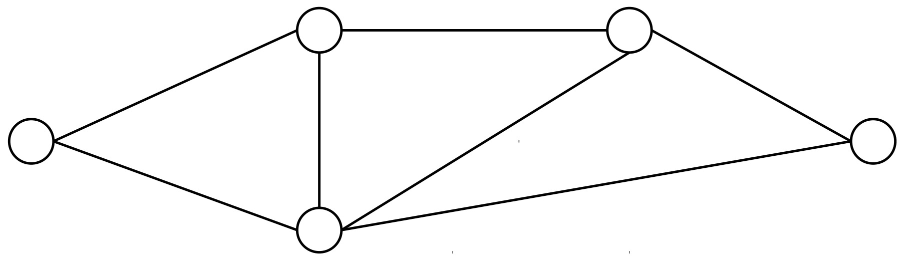

Figure 3: A typical mesh Network

DECNET and SNA can be modelled in this way.

A network protocol consists of a program for each network node. Each program consists of code and inputs as well as local state. The global state of the network consists of the local state of each node as well as the messages on network links. We define a *catastrophic* fault as a fault that arbitrarily corrupts the global network state, but not the program code or the inputs from outside the network.

Self-stabilization formalizes the following intuitive goal for networks: *despite a history of catastrophic failures, once catastrophic failures stop, the system should stabilize to correct behavior without manual intervention.* Thus self-stabilization is an abstraction of a strong fault-tolerance property for networks. It is an important property of real networks because:

- **Catastrophic faults occur:** Most network protocols are resilient to common failures such as nodes and link crashes but not to *memory corruption.* But memory corruption does happen from time to time. It is also hard to prevent a malfunctioning device from sending out an incorrect message.

- **Manual intervention has a high cost:** In a large decentralized network, restoring the network manually after a failure requires considerable coordination and expense. Thus even if catastrophic faults occur rarely, (say once a year) there is considerable incentive to make network protocols self-stabilizing. A reasonable guideline is that *the network should stabilize preferably before the user notices and at least before the user logs a service call!*

These issues are illustrated by the crash of the original ARPANET protocol ([Ros81] [Per83]). The protocol was carefully designed never to enter a state that contained three conflicting updates a, b, and c. Unfortunately, a malfunctioning node injected three such updates into the network and crashed. After this the network cycled continuously between the three updates. It took days of detective work [Ros81] before the problem was diagnosed. With hindsight, the problem could have been avoided by making the protocol self-stabilizing.

Self-stabilization is also attractive because a self-stabilizing program does not require initialization. The concept of an initial state makes perfect sense for a single sequential program. However, for a distributed program an initial state seems to be an artificial concept. How was the distributed program placed in such an initial state? Did this require another distributed program? Self-stabilization avoids these questions by eliminating the need for distributed initialization.

Probably the most exciting reason for self-stabilization is that can provide a *uniform approach towards fault-tolerance*, thus leading to simplification as well as strengthening of existing fault-tolerant protocols. This is because self-stabilization can subsume such common fault models as link and node failures. However, in order to so the self-stabilization recovery mechanisms must be fast enough to provide adequate response time for such common failures.

There appears to be a hierarchy of faults ranging from very rare faults like memory corruption (that occur at most once every few days) to fairly common faults like link and node crashes (that occur in the order of minutes) to very common faults like bit errors (that may occur every second). Thus it is adequate to recover from memory errors in the order of minutes, from link failures in the order of seconds, and from bit errors in the order of milliseconds. If the self-stabilization mechanism recovers too slowly (say in the order of minutes, as in [Per83]), then it is necessary to have separate, faster mechanisms to deal with common failures ([Per83]) like link and node failures.

On the other hand, if the self-stabilizing mechanisms recover in the order of seconds, then there is no need for separate mechanisms to deal with link and node failures. A good example of an existing self-stabilizing protocol that meets this criteria is the IEEE 802.1 bridge spanning tree protocol described in [Per85]. Clearly reducing the number of separate mechanisms leads to simpler protocols. It should be noted that it is unlikely that self-stabilization will be efficient enough to subsume the need for *all* other fault-recovery mechanisms; for example, most self-stabilizing protocols will probably need some retransmission scheme to deal with messages lost due to bit errors. However, the more mechanisms that can be subsumed, the simpler the resulting protocol.

1.3 Criticisms of Self-Stabilization

Despite the claims of the previous section, there are peculiar features of the self-stabilization model that need justification.

- The model allows network state to be corrupted but not program code. However, program code can be protected against arbitrary corruption of memory by redundancy since code is rarely modified. On the other hand, the state of a program is constantly being updated and it is not clear how one can prevent illegal operations on the memory by using checksums. It is even harder to prevent a malfunctioning node from sending out incorrect messages.

- The model only deals with catastrophic faults that *stop.* Byzantine models deal with continuous faults. However, in Byzantine models, only a fraction of nodes are allowed to exhibit arbitrary behavior. In the self-stabilization model, *all* nodes are permitted to start with arbitrary initial states. Thus, the two models are orthogonal and can even be combined.

- A self-stabilizing program P is is allowed to make initial mistakes. However, the important stabilizing protocols that we know of are used for routing, scheduling, and resource allocation tasks. For such tasks, initial errors only result in a temporary loss of service.

2 Local Checking and Correction: A Preview

To make the notion of local checking and correcting more concrete we quickly describe an example[2] of a protocol that works between two nodes, a sender node and a receiver node (Figure 4). The sender sends messages to the receiver who buffers the messages in a finite sized queue. Any message that arrives when the queue is full is dropped. In order to prevent unnecessary dropping of messages, a simple credit-based scheme can be employed.

The sender (Figure 4) keeps a credit register which stores the current credits available to the sender. Initially, assume that the receiver queue is empty and that the sender's credit register is equal to the size of the receiver queue (say Max). The sender only sends a message when the credit register is non-zero; after sending a message, the sender decrements its credit register by one. When the receiver removes a message from its queue, the receiver sends a CREDIT message back to the sender. When the sender receives such a CREDIT message, the sender increments its credit register by one.

It is easy to see that after proper initialization and assuming that no errors occur, no messages will be dropped. Under such conditions, the following condition (which we will later call a Local Predicate) holds at every instant. If at any instant we denote (see Figure 4) the value of the credit register by C, the number of messages in flight by M, the number of messages in the queue as Q and the number of credits in flight by CR, then it must be true that: $C + M + Q + CR = Max$. Intuitively, this can be seen by analogy to two banks that only transfer money between each other; assuming no errors, the total amount

[2] I added this example after Bob Simcoe presented our scheme in Aug 1993

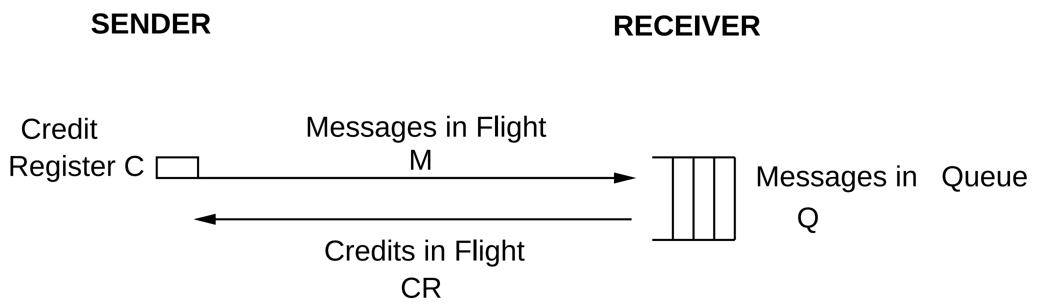

Figure 4: Credit Based Flow Control between a Sender and a Receiver

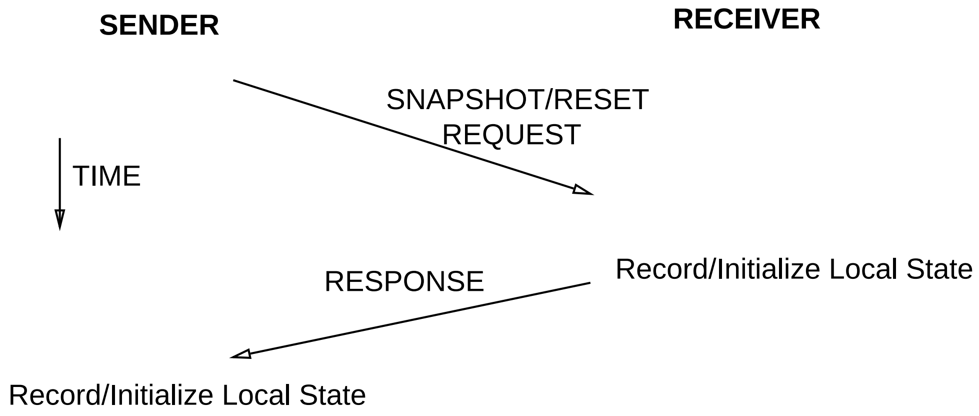

Figure 5: A Single Phase of Checking/Correction using Snapshots/Resets

of "money" (credits plus messages) in and between the two banks must be conserved. This local predicate ensures that there is always room in the queue for the messages in flight since $M + Q \leq Max$.

Unfortunately, this simple credit based scheme runs into trouble if the system is either improperly initialized or there are errors on the link. Link errors can result in lost or even (less likely) added credits. Credit loss can result in slowing down the sender and possibly even lead to deadlock; credit addition can lead to continuous dropping of messages.

We can make this protocol self-stabilizing by superimposing a periodic checking/correcting process (see Figure 5) on the original protocol. This process is triggered by a timer at the sender every few seconds. To initiate a checking phase, the sender (see Figure 5) sends a Snapshot ([CL85]) request message to the receiver. While checking, the sender also stops sending any messages on the link. When the receiver receives such a message, the receiver sends back a response containing the number of messages in its queue (Q) at the instant it sent the response. When the sender gets the response, the sender checks whether $Q + C = Max$, where C is the value of the credit register at the instant the response is received. If this condition is false, the sender infers that the local predicate is violated and initiates a reset phase.

To initiate a reset phase, the sender (see Figure 5) sends a Reset request message to the receiver. As in checking, the sender also stops sending any messages on the link until it gets a response. When the receiver receives such a request, the receiver empties its queue and sends back a response. When the sender gets the response, the sender reinitializes its credit register to *Max*. In other words, the receiver reinitializes its local state on sending the response and the sender reinitializes its local state on receiving the response. Its not hard to see that if no errors occur during the reset phase, the local predicate will hold at the end of the reset phase.

Several optimizations can be added to this basic scheme. For example, it is possible to avoid having the sender stop sending messages during a checking phase by keeping track of some extra state variables. For this particular example, it is also possible to avoid a separate reset phase; instead when the sender receives a snapshot response, the sender can locally correct the credit register to account for any discrepancy.

We can now illustrate the point made earlier that self-stabilization can subsume the need for other fault-tolerance mechanisms. If the checking/correction procedure is activated fairly often (doing it once every second on a high speed link requires negligible overhead), then *there is no need for separate mechanisms when either the sender or receiver nodes or the two links crash and recover.* For a crash and recovery we do nothing special. Clearly the local predicate can be violated by such actions. However, after the next checking and correction phase the flow control scheme will begin working correctly. In the interim, messages may be lost but this is comparable to the time most protocols take to reinitialize after a link recovers; during this period the protocol is not providing service to the user. Thus the final scheme is both simple and fault-tolerant.

This fault-tolerant credit based scheme was proposed by us (for use on high speed links) at Digital Equipment Corporation ([CSV89]). Recently, we proposed a variant of this scheme for hop-by-hop flow control on ATM[3] at the ATM Forum meeting in Aug 1993. Local checking and correction is practical!

2.1 Extending the Idea to a General Network

Briefly, the rest of this lecture can be described as an extension (of the simple link checking and correction scheme described in the last subsection) to general network protocols.

[3]ATM stands for Asynchronous Transfer Mode. In ATM, messages are fixed sized "cells". There can also be multiple "circuits" per link each of which must be independently flow-controlled and checked.

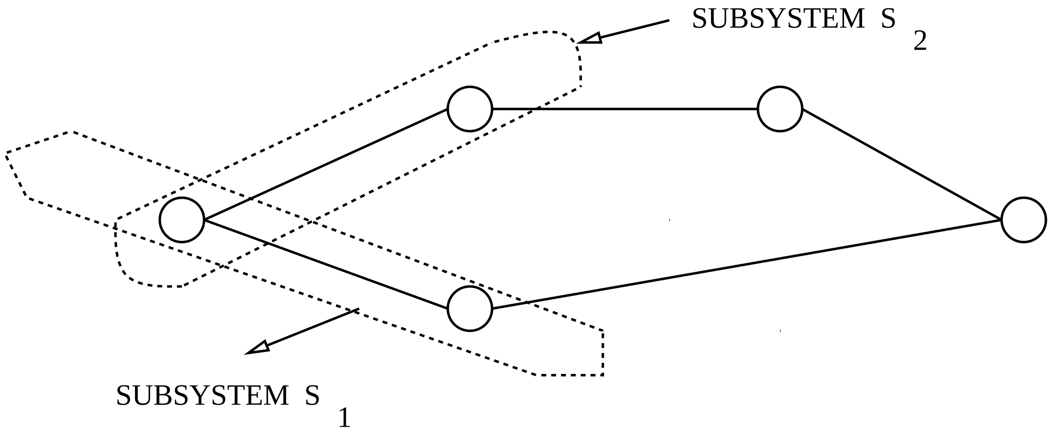

Figure 6: An Example of Two Overlapping Link Subsystems in a Network

So consider a network as shown in Figure 3. Recall that in the method of [KP90] there is a leader that periodically checks the network. If the leader discovers that the network is in an illegal state, the leader corrects the network by resetting it into a good state. Intuitively, *centralized* checking and correction is slow. It also has high message complexity.

Instead, we divide the network into a number of overlapping *link subsystems* as shown in Figure 6. A link subsystem consists of a pair of neighboring nodes and the channels between them. We wish to replace the *global, centralized* checking of [KP90] with *local, decentralized* checking. The intent, of course, is to allow each link subsystem to be checked in parallel. This results in faster stabilization.

We describe *sufficient* conditions under which these methods can be applied. Intuitively, a network protocol is *locally checkable* if whenever the protocol is in a bad state, some link subsystem is also in a bad state. Thus if the protocol is in a bad state, some link subsystem will be able to detect this fact locally. As in [KP90], we can correct a locally checkable protocol by doing (what we call) *global correction* of the network. However, in some cases we can do even better if the protocol is also *locally correctable.*

Intuitively, a network protocol is locally correctable if the network can be corrected to a good state by each link subsystem independently correcting itself to a good state. Clearly, this is non-trivial because link subsystems overlap (see Figure 6) at nodes. In the figure, the correction of link subsystem S_1 may cause subsystem S_2 to become incorrect.

It is perhaps surprising that a number of useful network protocols are both locally checkable and locally correctable. We have designed locally checkable and correctable protocols for mutual exclusion, network resets, and end-to-end communication across an unreliable network. Both the end-to-end and reset protocols work on arbitrary topologies.

Protocols that are locally checkable but work on a tree topology can be stabilized in time proportional to the height of the tree. Thus we can remove the need for local correctability if the underlying topology is a tree. Another way to remove the need for local correctability (without restricting the topology to a tree) is to pay a price in stabilization time. Protocols that are locally checkable but not locally correctable can be made self-stabilizing by doing *global* correction using the network reset protocol developed in this thesis. The price for using global correction is that the stabilization time now becomes proportional to the number of nodes. We describe stabilizing spanning tree and topology update protocols that use local checking and global correction.

Thus while local checking cannot be used to solve every problem, there are a large number of useful protocols that can be efficiently stabilized using this notion.

3 Definitions of Stabilization

3.1 Execution Definitions

All the existing definitions of stabilization are in terms of the states and executions of a system. We will begin with a definition of stabilization that corresponds to the standard definitions (for example, that of Katz and Perry [KP90]). Next, we will describe another definition of stabilization in terms of external behaviors. We believe that the definition of behavior stabilization is appropriate for large systems that require modular proofs. However, the definition of execution stabilization given below is essential in order to *prove* results about behavior stabilization.

Recall that the I/O Automaton model we are using in this class models a protocol as an automaton (state machine) with a few important features. First, transitions of the automaton are labeled with action names, and all actions are classified into three categories: input, output, and internal. Intuitively, *input actions* are actions caused by the external world or environment and to which the automaton must respond. *Output actions* are actions caused by the automaton and to which the environment must respond. Finally, *internal actions* are transitions that are neither input or output actions: such actions only change the internal state of the automaton.

Recall also that when an automaton "runs" it produces an *execution*, an alternating sequence of states and actions that represent a sequence of state transitions. The external behavior corresponding to an execution a is the subsequence of a consisting only of input and output actions.

Suppose we define the correctness of an automaton in terms of a set C of legal executions.

For example, for a token passing system, we can define the legal executions to be those in which there is exactly one token in every state, and in which every process periodically receives a token.

What do we mean when we say that an automaton A stabilizes to the executions in set C. Intuitively, we mean that eventually all executions of A begin to "look like" an execution in set C. For example, suppose C is the set of legal executions of a token passing system. Then in the initial state of A there may be zero or more tokens. However, the definition requires that eventually there is some suffix of any execution of A in which there is exactly one token in any state.

Formally:

Definition 1 *Let C be a set of executions. We say that automaton A stabilizes to the executions in C if for every execution α of A there is some suffix of execution α that is in C.*

We can extend this definition in the natural way to define what it means for an automaton A to stabilize to the executions of another automaton B.

3.2 Definitions of Stabilization based on External Behavior

In the I/O Automaton model, the correctness of an automaton is specified in terms of its external behaviors. Recall that the external behavior corresponding to an execution a is the subsequence of a consisting of input and output actions. Thus we specify the correctness of a token passing system without any reference to the state of the system. We can do so by specifying the ways in which token delivery and token return actions (to and from some external users of the token system) can be interleaved.

Thus it natural to look for a definition of stabilization in terms of external behaviors. We would also hope that such a definition would allow us to modularly "compose" results about the stabilization of parts of a system to yield stabilization results about the whole system.

Now an IOA A is said to *solve* a problem P if the behaviors of A are contained in P. For stabilization, however, it is reasonable to weaken this definition and ask only that an IOA *eventually* exhibit correct behavior. Formally:

Definition 2 *Let P be a problem (i.e, a set of behaviors). An IOA A stabilizes to the behaviors in P if for every behavior β of A there is a suffix of β that is in P.*

Similarly, we can specify that A stabilizes to the behaviors of some other automaton B in the same way. The behavior stabilization definition used in this section and [Var92b] is a special case of a definition suggested by Nancy Lynch.

Finally, we have a simple lemma that ties together the execution and behavior stabilization definitions. It states that execution stabilization implies behavior stabilization. In fact, the only method we know to *prove* a behavior stabilization result is to first prove a corresponding execution stabilization result, and then use this lemma. Thus the behavior and execution stabilization definitions complement each other in this lecture: the former is typically used for *specification* and the latter is often used for *proofs*.

Lemma 1 *If IOA A stabilizes to the executions of IOA B then IOA A stabilizes to the behaviors of B.*

3.3 Discussion on the Stabilization Definitions

First, notice that we have defined what it means for an arbitrary IOA to stabilize to some target set or automaton. Typically, we will be interested in proving stabilization properties only for a special kind of automata: unrestricted automata. An unrestricted IOA (UIOA) is one in which all states of the IOA are also start states. Such an IOA models a system that has been placed in an arbitrary initial state by an arbitrary initial fault.

A possible modification (which is sometimes needed) is to only require (in Definitions 2 and 1) that the suffix of a behavior (execution) be a *suffix* of a behavior (execution) of the target set. One problem with this modified definition is that we know of no good proof technique to prove that the behaviors (executions) of an automaton are *suffixes* of a specified set of behaviors (executions).By contrast, it is much easier to prove that every behavior of an automaton has a suffix that is in a specified set. Thus we prefer to use the simpler definitions for what follows.

4 Examples from Dijkstra's Shared Memory Model

In Dijkstra's [Dij74] model, a network protocol is modelled using a graph of finite state machines. In a single move, a *single* node is allowed to read the state of its neighbors, compute, and then possibly change its state. In a real distributed system such atomic communication is impossible. Typically communication has to proceed through channels. While Dijkstra's original model is not very realistic, it is probably the simplest model of an asynchronous distributed system. This simple model provided an ideal vehicle for *introducing* [Dij74] the concept of stabilization without undue complexity. For this section, we will use Dijkstra's original model to *introduce* two important and general techniques for self-stabilization: *local checking and correction* and *counter flushing*. In the next section, we will introduce a more realistic message passing model.

4.1 Dijkstra's Shared Memory Model and IOA

How can we map Dijkstra's model into the IOA model? Suppose each node in Dijkstra's model is a separate automaton. Then in the Input/Output automata model, it is not possible to model the simultaneous reading of the state of neighboring nodes. The solution we use is to dispense with modularity and model *the entire network as a single automaton.* All actions, such as reading the state of neighbors and computing, are *internal actions.* The asynchrony in the system, which Dijkstra modelled using a "demon", is naturally a part of the IOA model. Also, we will describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

Formally:

A *shared memory network automaton* $\mathcal{N}$ for graph $G = (E, V)$ is an automaton in which:

- The state of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, one for each node $u \in V$. For any state s of $\mathcal{N}$, we use $s|u$ to denote s projected onto S_u. This is also read as the state of node u in global state s.

- All actions of $\mathcal{N}$ are internal actions and are partitioned into sets, $A_u(\mathcal{N})$, one for each node $u \in V$

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Consider any state s' of $\mathcal{N}$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u. Then there is some transition $(s', \pi, \tilde{s}')$ of $\mathcal{N}$ such that $\tilde{s}'|v = \tilde{s}|v$ for u and all u's neighbors in G.

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Then $s|v = \tilde{s}|v$ for all $v \neq u$.

Informally, the third condition requires that the transitions of a node $u \in V$ only depend on the state of node u and the states of of the neighbors of u in G. The fourth condition requires that the effect of a transition assigned to node $u \in V$ can only be to change the state of u.

4.2 Dijkstra's Second Example as Local Checking and Correction

In this section, we will begin by reconsidering the second example in [Dij74]. The derivation here is based on some unpublished work I did with Anish Arora and Mohamed Gouda at the University of Texas. This protocol is essentially a token passing protocol on a line of nodes with process indices ranging from 0 to $n-1$. Imagine that the line is drawn vertically so that process 0 is at the bottom of the line (and hence is called "bottom") and Process $n-1$

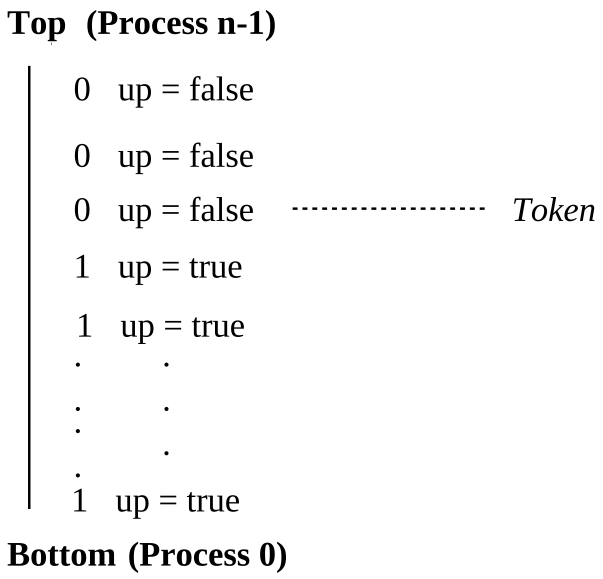

Figure 7: Dijktra's protocol for token passing on a line

is at the top of the line (and called "top"). This is shown in Figure 7. The down neighbor of Process i is Process $i-1$ and the up neighbor is Process $i+1$. Process $n-1$ and Process 0 are not connected.

Dijkstra observed that it is impossible (without randomization) to solve mutual exclusion in a stabilizing fashion if all processes have identical code. To break symmetry, he made the code for the "top" and "bottom" processes different from the code for the others.

Dijkstra's second example is modelled by the automaton $D2$ shown in Figure 8. Each process i has a boolean variable up_i, and a bit x_i. Roughly, up_i is a pointer at node i that points in the direction of the token, and x_i is a bit that is used to implement token passing. Figure 7 shows a state of this protocol when it is working correctly. First, there can be at most two consecutive nodes whose *up* pointers differ in value and the token is at one of these two nodes. If the two bits at the two nodes are different (as in the figure) then the token is at the upper node; else the token is at the lower node.

For the present, assume that all processes start with $x_i = 0$. Also, initially assume that $up_i = \mathit{false}$ for all processes other than process 0. We will remove the need for such initialization below. We start by understanding the correct executions of this protocol when it has been correctly initialized.

A process i is said to have the token when any action at Process i is enabled. As usual the system is correct when there is at most one token in the system. Now, it is easy to see that in the initial state only $\textsc{Move\_Up}_0$ is enabled. Once node 0 makes a move, then $\textsc{Move\_Up}_1$ is enabled followed by $\textsc{Move\_Up}_2$ and so on as the "token" travels up the line. Finally the

The state of the system consists of a boolean variable
up_i and a bit x_i, one for every process in the line.
We will assume that $up_0 = \mathit{true}$ and $up_{n-1} = \mathit{false}$ by definition

In the initial state $x_i = 0$ for $i = 0 \ldots n-1$ and $up_i = \mathit{false}$ for $i = 1 \ldots n-1$

MOVE_UP$_0$ (*action for the bottom process only to move token up*)
Precondition: $x_0 = x_1$ and $up_1 = \mathit{false}$
Effect: $x_0 := \sim x_0$

MOVE_DOWN$_{n-1}$ (*action for top process only to move token down*)
Precondition: $x_{n-2} \neq x_{n-1}$
Effects:
$x_{n-1} := x_{n-2}$;

MOVE_UP$_i, 1 \leq i \leq n-2$ (*action for other processes to move token up*)
Precondition: $x_i \neq x_{i-1}$
Effects:
$x_i := x_{i-1}$;
$up_i := \mathit{true}$; (*point upwards in direction token was passed*)

MOVE_DOWN$_i, 1 \leq i \leq n-2$ (*action for other processes to move token down*)
Precondition: $x_i = x_{i+1}$ and $up_i = \mathit{true}$ and $up_{i+1} = \mathit{false}$
Effect: $up_i := \mathit{false}$; (*point downwards in direction token was passed*)

All actions are in a separate class.

Figure 8: Automaton $D2$: a version of Dijkstra's second example with initial states. The protocol does token passing on a line using nodes with at most 4 states.

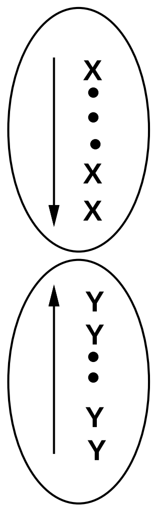

Figure 9: In the good states for Dijktra's second example, the line can be partitioned into 2 bands with the token at the boundary.

token reaches node $n-1$, and we reach a state s in which $x_i = x_{i+1}$ for $i = 0 \ldots n-3$ and $x_{n-1} \neq x_{n-2}$. Also in state s, $up_i = true$ for $i = 0 \ldots n-2$ and $up_{n-1} = false$. Thus in state s, MOVE_DOWN$_{n-1}$ is enabled and the token begins to move down the line by executing MOVE_DOWN$_{n-2}$ followed by MOVE_DOWN$_{n-3}$ and so on until we reach the initial state again. Then the cycle continues. Thus in correct executions, the "token" is passed up and down the line.

In the good states for Dijktra's second example, the line can be partitioned into two bands as shown in Figure 9. All bit and pointer values within a band are equal. (If a node within a band has $up_i = false$ we sketch it as a pointer that points downwards). All nodes within the upper band point downwards and all nodes within the lower band point upward. The token is at the boundary between the two bands. If the bit value X of the upper band is equal to the bit value Y of the lower band, the token is moving downwards; if the two bit values are unequal the token is moving downwards.

We describe these "good states" of $D2$ (that occur in correct executions) in terms of local predicates. In the shared memory model, a local predicate is any predicate that only refers to the state variables of a pair of neighbors. Intuitively, we see that if two neighboring nodes have the same pointer value, then their bits are equal; also if a node is pointing upwards, then so is its lower neighbor. Thus in a good state of $D2$, two properties are true for any Process i other than 0:

- If $up_{i-1} = up_i$ then $x_{i-1} = x_i$.
- If $up_i = true$ then $up_{i-1} = true$.

First, we prove that if these two local predicates hold for all $i = 1 \ldots n-1$, then there is exactly one action enabled. Intuitively, since $up_{n-1} = false$ and $up_0 = true$, we can start with process $n-1$ and go down the line until we find a pair of nodes i and $i-1$ such that $up_i = false$ and $up_{i-1} = true$. Consider the first such pair. Then the second predicate guarantees us that there is exactly one such pair. The first predicate then guarantees that all nodes $j < i-1$ have $x_j = x_{i-1}$ and all nodes $k > i$ have $x_k = x_i$. Thus only one action is enabled. If $x_i = x_{i-1}$ and $i-1 \neq 0$ then only MOVE_DOWN$_{i-1}$ is enabled. If $x_i = x_{i-1}$ and $i-1 = 0$ then only MOVE_UP$_0$ is enabled. If $x_i \neq x_{i-1}$ and $i \neq n-1$ then only MOVE_UP$_i$ is enabled. If $x_i \neq x_{i-1}$ and $i = n-1$ then only MOVE_DOWN$_{n-1}$ is enabled.

Similarly we can show that if exactly one action is enabled then all local predicates hold. Thus if the system is in a bad state, some pair of neighbors will be able to detect this fact. Hence we conclude that $D2$ is *locally checkable*. However, we would like to go even further and be able to correct the system to a good state by adding extra correction actions to each node. If we can add correction actions so that all link predicates will eventually become true, then we say that the $D2$ is also *locally correctable*. Adding local correction actions is tricky because the correction actions of nodes may "interfere" and result in a form of "thrashing".

However, a line is a special case of a tree with (say) 0 as the root; each node i other than 0, can consider $i-1$ to be its parent. The tree topology suggests a simple correction strategy. For each node $i \neq 0$, we can add a new action CORRECT_CHILD$_i$ (which is a separate class for each i). Basically, CORRECT_CHILD$_i$ checks whether the link predicate on the link between i and its parent is true. If not, i changes its state such that the predicate becomes true. Notice that CORRECT_CHILD$_i$ leaves the state of i's parent unchanged. Suppose j is the parent of i and k is the parent of j. Then CORRECT_CHILD$_i$ will leave the local predicate on link (j, k) true if it was true in the previous state.

Thus we have an important stability property: correcting a link does not affect the correctness of links above it in the tree. Using this it is easy to see that eventually all links will be in a good state and so the system is in a good state. We will ignore the details of this proof but the basic idea should be clear. Dijkstra's actual code does not use an explicit correction action. However, we feel that this way of understanding his example is clearer and illustrates a general method. The general method of local checking and correction was first introduced in [APV91b].

4.3 Dijkstra's first example as Counter Flushing

The counter flushing paradigm described in this section is taken from [Var92b].

Dijkstra's first example is modelled by the automaton $D2$ shown in Figure 10. As in the previous example, the nodes (once again numbered from 0 to $n-1$) are arranged such

The state of the system consists of an integer variable
$count_i \in \{0, \ldots n\}$, one for every process in the ring.
We assume that Process 0 and $n-1$ are neighbors

In the initial state $count_i = 0$ for $i = 1 \ldots n-1$ and $count_1 = 1$

MOVE_0 (*action for Process 0 only *)
Precondition: $count_0 = count_{n-1}$ (*equal to anticlockwise neighbor?*)
Effect: $count_0 := (count_0 + 1) \bmod (n+1)$ (*increment counter*)

$\text{MOVE}_i, 1 \leq i \leq n-1$ (*action for other processes*)
Precondition: $count_i \neq count_{i-1}$ (*not equal to anticlockwise neighbor?*)
Effects:
$count_i := count_{i-1}$;(*set equal to anticlockwise neighbor*)

All actions are in a separate class

Figure 10: Automaton $D1$: a version of Dijkstra's first example with initial states. The protocol does token passing on a ring using nodes with n states.

that node 1 has node 0 and node 2 as its neighbors and so on. However, in this case we also assume that Process 0 and $n-1$ are neighbors. In other words, by making 0 and $n-1$ adjacent we have made the line into a ring. For process i, let us call Process $i-1$ (we assume that all arithmetic on indices and counters is mod n) the anticlockwise neighbor of i and $i+1$ the clockwise neighbor of i.

Each node has a counter $count_i$ in the range $0, \ldots n$ that is incremented mod $n+1$. Once again the easiest way to understand this protocol is to understand what happens when it is properly initialized. Thus assume that initially Process 0 has its counter set to 1 while all other processes have their counter set to 0. Processes other than 0 are only allowed to "move" (see Figure 10) when their counter differs in value from that of their anticlockwise neighbor; in this case, the process is allowed to make a move by setting its counter to equal that of its anticlockwise neighbor. Thus initially, only Process 1 can make a move after which Process 1 has its counter equal to 1; next, only Process 2 can move, after which Process 2 sets its counter equal to 1; and so on, until the value 1 moves clockwise around the ring until all processes have their counter equal to 1.

Process 0 on the other hand cannot make a move until Process $n-1$ has the same counter

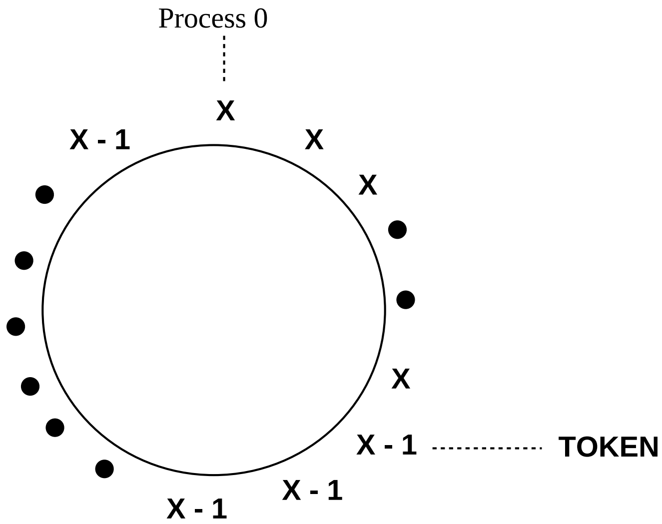

Figure 11: In the good states for Dijktra's first example, the ring can be partitioned into 2 bands with the token at the boundary.

value as Process 0. Thus until Process 1 sets its counter to 1, Process 0 cannot make a move. However, when this happens, Process 1 increments its counter mod $n+1$. Then the cycle repeats as now the value 2 begins to move across the ring (assuming $n > 2$) and so on. Thus after proper initialization, this system does perform a form of token passing on a ring; each node is again considered to have the token, when the system is in a state in which the node can take a move.

The good global states of the protocol can be sketched in Figure 11 Notice that the ring can be partitioned into 2 bands. All counter values within a band are equal and the band that includes the top node has a counter value one higher than the lower band. The token is at the boundary between the two bands; after that node makes a move, the top band becomes larger and the lower band becomes smaller.

It is easy to see that the system is in a good state iff the following local predicates are true.

- For $i = 1 \ldots n-1$, either $count_{i-1} = count_i$ or $count_{i-1} = count_i + 1$.
- Either $count_0 = count_{n-1}$ or $count_0 = count_{n-1} + 1$.

The system is locally checkable but it does not appear to be locally correctable. However, it does stabilize using a paradigm that we can call *counter flushing.* Even if the counter values are arbitrarily initialized (in the range $0, \ldots, n$) the system will eventually begin executing as some suffix of a properly initialized execution. We will prove this informally using three claims:

- **In any execution, Process 0 will eventually increment its counter.** Suppose not. Then since Process 0 is the only process that can "produce" new counter values, the number of distinct counter values cannot increase. If there are two or more distinct counter values, then moves by Processes other than 0 will reduce the number of distinct counter values to 1, after which Process 0 will increment its counter.

- **In any execution, Process 0 will eventually reach a "fresh" counter value that is not equal to the counter values of any other process.** To see this, note that that in the initial state there are at most n distinct counter values. Thus there is some counter value say m that is not present in the initial state. Since, process 0 keeps incrementing its counter, Process 0 will eventually reach m and in the interim no other process can set their counter value to m.

- **Any state in which Process 0 has a fresh counter value m is eventually followed by a state in which all processes have counter value m.** It is easy to see that the value m moves clockwise around the ring "flushing" any other counter values, while Process 0 remains at m. This is why I call this paradigm counter flushing.

The net effect is that any execution of $D1$ eventually reaches a good state in which it remains. The general counter flushing paradigm can be stated roughly as follows:

- A sender (in our case, Process 0) periodically sends a message to a set of responders; after all responders have received the message the sender sends the next message (this corresponds to one cycle around the ring in our case).

- To make the protocol stabilizing, the sender numbers each message with a counter. The size of the counter must be greater than the maximum number of distinct counter values that can be present in the network in any state. The sender increments its counter after the message has reached all responders.

- The protocol must ensure that the sender counter value will always increment and so eventually reach a "fresh" value not present in the network. A freshly numbered message m from the sender must guarantee that all old counter values will be "flushed" from the network before the sender sends the next message.

In Dijkstra's first example, the flushing property is guaranteed because the topology consists of a ring of unidirectional links. Similar forms of counter flushing can be used to implement Data Links ([AB89]) and token passing [DIM91]) between a pair of nodes. Counter flushing is, however, not limited to rings or pair of nodes. Katz and Perry [KP90] extend the use of counter flushing to arbitrary networks in an ingenious way. The stabilizing end-to-end protocol ([APV91b]) is obtained by first applying local correction to the Slide protocol [AGR92] and then applying a variant of counter flushing to the Majority protocol of [AGR92].

5 Message Passing Model

Having understood the techniques of counter flushing and local checking and correction in a shared memory model, we move to a more realistic message passing model. The work in the next three sections is based on [Var92b], which formalizes the ideas introduced in [APV91b]. To model a network protocol, we will model the network topology, the links between nodes, and the nodes themselves. Our model is essentially the standard asynchronous message passing model except for two major differences:

- The major difference is that links are restricted to store at most one packet at a time.
- We assume that for every pair of neighbors, there is some *a priori* way of assigning one of the two nodes as the "leader" of the pair.

We will argue that even with these differences our model can be implemented in real networks.

5.1 Modelling the Topology

We will call a directed graph (V, E) *symmetric* if for every edge $(u, v) \in E$ there is an edge $(v, u) \in E$.

Definition 3 *A* topology graph $G = (V, E, l)$ *is a symmetric, directed graph* (V, E) *together with a leader function* l *such that for every edge* $(u, v) \in E$*,* $l(u, v) = l(v, u)$ *and either* $l(u, v) = u$ *or* $l(u, v) = v$*.*

We use $E(G)$ and $V(G)$ to denote the set of edges and nodes in G. If it is clear what graph we mean we sometimes simply say E and V. As usual, if $(u, v) \in E$ we will call v a neighbor of u.

5.2 Modelling Links

Traditional models of a data link have used what we call *Unbounded Storage* Data Links that can store an unbounded number of packets. Now, real physical links do have bounds on the number of stored packets. However, the unbounded storage model is a useful abstraction in a non-stabilizing context.

Unfortunately, this is no longer true in a stabilizing setting. *If the link can store an unbounded number of packets, it can have an unbounded number of "bad" packets in the initial state.* It has been shown [DIM91] that almost any non-trivial task is impossible in such a setting. Thus in a stabilizing setting it is necessary to define Data Links that have bounded storage.

A *network automaton* for topology graph G consists of a node automaton for every vertex in G and one channel automaton for every edge in G. We will restrict ourselves to a special type of channel automaton, a unit storage data link or UDL for short. Intuitively, a UDL can only store at most one packet at any instant. Node automata communicate by sending packets to the UDLs that connect them. In the next section, we will argue that a UDL can be implemented over real physical channels.

We fix a packet alphabet P. We assume that $P = P_{data} \cup P_{control}$ consists of two disjoint packet alphabets. These correspond to what we call data packets and control packets. The specification for a UDL will allow both data and control packets to be sent on a UDL.

Definition 4 *We say that $C_{u,v}$ is the UDL corresponding to ordered pair (u, v) if $C_{u,v}$ is the IOA defined in Figure 12.*

$C_{u,v}$ is a UIOA since we have not defined any start states for $C_{u,v}$. The external interface to $C_{u,v}$ includes an action to send a packet at node u ($\text{SEND}_{u,v}(p)$), an action to receive a packet at node v ($\text{RECEIVE}_{u,v}(p)$), and an action $\text{FREE}_{u,v}$ to tell the sender that the link is ready to accept a new packet. The state of $C_{u,v}$ is simply a single variable $Q_{u,v}$ that stores a packet or has the default value of *nil*.

Notice two points about the specification of a UDL. The first is that if the UDL has a packet stored, then any new packet sent will be dropped. Second, the FREE action is enabled continuously whenever the UDL does not contain a packet.

5.3 Modelling Network Nodes and Networks

Next we specify node automata. We do so using a set that contains a node automaton for every node in the topology graph. For every edge incident to a node u, a node automaton N_u must have interfaces to send and receive packets on the channels corresponding to that edge. However, we will go further and require that nodes obey a certain stylized convention in order to receive feedback from and send packets on links.

In the specification for a UDL if a packet p is sent when the UDL already has a packet stored, then the new packet p is dropped. We will prevent packets from being dropped by requiring that the sending node keep a corresponding *free* variable for the link that records

Each p belongs to the packet alphabet P defined above.

The state of the automaton consists of a single variable $Q_{u,v} \in P \cup nil$.

$\text{SEND}_{u,v}(p)$ (*input action*)
Effect:
If $Q_{u,v} = nil$ then $Q_{u,v} := p$;

$\text{FREE}_{u,v}$ (*output action*)
Precondition: $Q_{u,v} = nil$
Effect: None

$\text{RECEIVE}_{u,v}(p)$ (*output action*)
Precondition: $p = Q_{u,v} \neq nil$
Effect: $Q_{u,v} := nil$;

The FREE and RECEIVE actions are in separate classes

Figure 12: Unit Storage Data Link automaton

whether or not the link is free to accept new packets. The sender sets the *free* variable to *true* whenever it receives a FREE action from the link. We require that the sender only send packets on the link when the *free* variable is *true*. Finally, whenever the sender sends a packet on the link, the sender sets its *free* variable to *false*.

We wish the interface to a UDL to stabilize to "good behavior" even when the sender and link begin in arbitrary states. Suppose the sender and the link begin in arbitrary states. Then we can have two possible problems. First, if $free = true$ but the UDL contains a packet, then the first packet sent by the sender can be dropped. However, it is easy to see that all subsequent packets will be accepted and delivered by the link. This is because after the first packet is sent, the sender will never set *free* to *true* unless it receives a FREE notification from the link. But a FREE notification is delivered to the sender only when the link is empty. The second possible problem is deadlock. Suppose that initially $free = false$ but the channel does not contain a packet. To avoid deadlock, the UDL specification ensures that the FREE action is enabled continuously whenever the link does not contain a packet.

A formal description of node automata can be found in [Var92b]. We omit it here. Finally a network automaton is the composition of a set of node and channel automata.

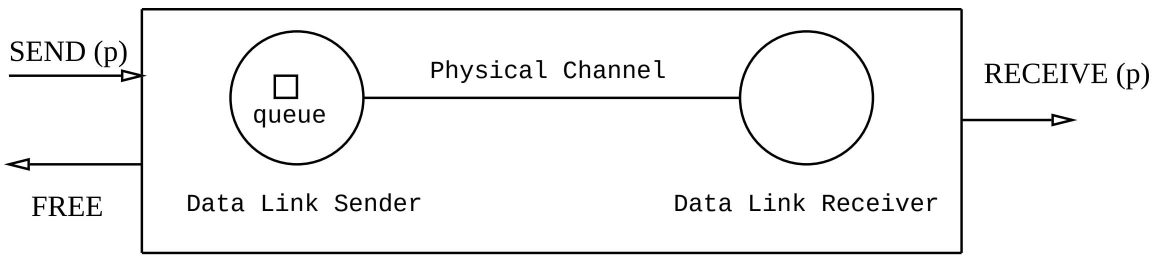

Figure 13: Implementing a UDL over a physical channel

5.4 Implementing the Model in Real Networks

In a real network implementation, the physical channel connecting any two neighboring nodes would typically not be a UDL. For example, a telephone line connecting two nodes can often store more than one packet. The physical channel may also not deliver a free signal. Instead, an implementation can *construct* a Data Link protocol on top of the physical channel such that the resulting Data Link protocol *stabilizes* to the behaviors of a UDL (e.g. [AB89], [Spi88a]).

Figure 13 shows the structure of such a Data Link protocol over a physical link. The sender end of the Data Link protocol has a queue that can contain a single packet. When the queue is empty, the FREE signal is enabled. When a SEND(p) arrives and the queue is empty, p is placed on the queue; if the queue is full, p is dropped. If there is a packet on the queue, the sender end constantly attempts to send the packet. When the receiving end of the Data Link receives a packet, the receiver sends an ack to the sender. When the sender receives an ack for the packet currently in the queue, the sender removes the packet from the queue.

If the physical channel is initially empty and the physical channel is FIFO (i.e., does not permute the order of packets), then a standard stop and wait or alternating bit protocol [BSW69] will implement a UDL. However, if the physical channel can initially store packets, then the alternating bit protocol is not stabilizing [Spi88a]. There are two approaches to creating a stabilizing stop and wait protocol. Suppose the physical channel can store at most X packets in both directions. Then [AB89] suggest numbering packets using a counter that has at least $X+1$ values. Suppose instead that no packet can remain on the physical channel for more than a bounded amount of time. [Spi88a] exploits such a time bound to build a stabilizing Data Link protocol. The main idea is to use either numbered packets (counter flushing) or timers (timer flushing) to "flush" the physical channel of stale packets.

A stop and wait protocol is not very efficient over physical channels that have a high transmission speed and/or high latency. It is easy to generalize a UDL to a *Bounded Storage Data Link* or BDL that can store more than one packet.

6 Local Checking and Correction in our Message Passing Model

In this section, we introduce formal definitions of local checkability and local correctability. The definitions for a message passing model will be slightly more complex than corresponding ones for shared memory model because of the presence of channels between nodes.

6.1 Link Subsystems and Local Predicates

Consider a network automaton with graph G. Roughly speaking, a property is said to be local to a subgraph G' of G if the truth of the property can be ascertained by examining only the components specified by G'. We will concentrate on *link subsystems* that consist of a pair of neighboring nodes u and v and the channels between them. It is possible to generalize our methods to arbitrary subsystems.

In the following definitions, we fix a network automaton $\mathcal{N} = Net(G, N)$.

Definition 5 *We define the (u,v) link subsystem of $\mathcal{N}$ as the composition of N_u, $C_{u,v}$, $C_{v,u}$, and N_v.*

For any state s of $\mathcal{N}$: $s|u$ denotes s projected on to node N_u and $s|(u,v)$ denotes s projected onto $C_{u,v}$. Thus when $\mathcal{N}$ is in state s, the state of the (u,v) subsystem is the 4-tuple: $(s|u, s|(u,v), s|(v,u), s|v)$.

A predicate L of $\mathcal{N}$ is a subset of the states of $\mathcal{N}$. Let (u,v) be some edge in graph G of $\mathcal{N}$. A *local predicate* $L_{u,v}$ of $\mathcal{N}$ for edge (u,v) is a subset of the states of the (u,v) subsystem in $\mathcal{N}$. We use the word "local" because $L_{u,v}$ is defined in terms of the (u,v) subsystem.

The following definition provides a useful abbreviation. It describes what it means for a local property to hold in a state s of the entire automaton.

Definition 6 *We say that a state s of $\mathcal{N}$* satisfies *a local predicate $L_{u,v}$ of $\mathcal{N}$ iff $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$.*

We will make frequent use of the concept of a closed predicate. Intuitively, a property is closed if it remains true once it becomes true. In terms of local predicates:

Definition 7 *A local predicate $L_{u,v}$ of network automaton $\mathcal{N}$ is closed if for all transitions $(s, \pi, \tilde{s})$ of $\mathcal{N}$, if s satisfies $L_{u,v}$ then so does $\tilde{s}$.*

The following definitions provide two more useful abbreviations. The first gives a name to a collection of local predicates, one for each edge in the graph. The second, the conjunction of a collection of "local properties", is the property that is true when all local properties hold at the same time. We will require that the conjunction of the local properties is non-trivial – i.e., there is some global state that satisfies all the local properties.

Definition 8 *$\mathcal{L}$ is a* link predicate set *for $\mathcal{N} = Net(G, N)$ if for each $(u, v) \in G$ there is some $L_{u,v}$ such that:*

- *If $(a, b, c, d) \in L_{u,v}$ then $(d, c, b, a) \in L_{v,u}$. (i.e., $L_{u,v}$ and $L_{v,u}$ are identical except for the way the states are written down.)*
- $\mathcal{L} = \{L_{u,v}, (u, v) \in G\}$
- *There is at least one state s of $\mathcal{N}$ such that s satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}$.*

Definition 9 *The conjunction of a link predicate set $\mathcal{L}$ is the predicate $\{s : s$ satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}\}$. We use $Conj(\mathcal{L})$ to denote the conjunction of $\mathcal{L}$.*

Note that $Conj(\mathcal{L})$ cannot be the null set by the definition of a link predicate set.

6.2 Local Checkability

Suppose we wish a network automaton $\mathcal{N}$ to satisfy some property. An example would be the property "all nodes have the same color". We can often specify a property of $\mathcal{N}$ formally using a predicate L of $\mathcal{N}$. Intuitively, $\mathcal{N}$ can be locally checked for L if we can ascertain whether L holds by checking all link subsystems of $\mathcal{N}$. The motivation for introducing this notion is performance: in a distributed system we can check all link subsystems in parallel in constant time. We formalize the intuitive notion of a locally checkable property as follows.

Definition 10 *A network automaton $\mathcal{N}$ is locally checkable for predicate L using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{L}$ is a link predicate set for $\mathcal{N}$ and $L \supseteq Conj(\mathcal{L})$.*
- *Each $L_{u,v} \in \mathcal{L}$ is closed.*

The first item in the definition requires that L holds if a collection of local properties all hold. The second item is perhaps more surprising. It requires that each local property also be closed.

We add this extra requirement because in an asynchronous distributed system it appears to be impossible to check whether an arbitrary local predicate holds *all* the time. What we can do is to "sample" the local subsystem periodically to see whether the local property holds. Suppose the network automaton consists of three nodes u, v and w and such that v is the neighbor of both u and w. Suppose the property L that we wish to check is the conjunction of two local predicates $L_{u,v}$ and $L_{v,w}$. Suppose further that exactly one of the two predicates is always false, and the predicate that is false is constantly changing. Then whenever we "check" the (u, v) subsystem we might find $L_{u,v}$ true. Similarly whenever we "check" the (v, w) subsystem we might find $L_{v,w}$ true. Then we may never detect the fact that L does not hold in this execution. We avoid this problem by requiring that $L_{u,v}$ and $L_{v,w}$ be closed.

6.3 Local Correctability

The motivation behind local checking was to efficiently ensure that some property L holds for network automaton $\mathcal{N}$. We would also like to *efficiently* correct $\mathcal{N}$ to make the property true. We have already set up some plausible conditions for local checking. Can we find some plausible conditions under which $\mathcal{N}$ can be *locally corrected*?

To this end we define a local reset function f. This is a function with three arguments: the first argument is a node say u, the second argument is any state of node automaton N_u, and the second argument is a neighbor v of u. The function produces a state of the node automaton corresponding to the first argument. Let s be a state of $\mathcal{N}$; recall that $s|u$ is the state of N_u. Then $f(u, s|u, v)$ is the state of N_u obtained by applying the local reset function at u with respect to neighbor v. We will abuse notation by omitting the first argument when it is clear what the first argument is. Thus we prefer to write $f(s|u, v)$ instead of the more cumbersome $f(u, s|u, v)$.

We will insist that f meet two requirements so that f can be used for local correction (Definition 11).

Assume that the property L holds if a local property $L_{u,v}$ holds for every edge (u, v). The first requirement is that if any (u, v) subsystem does not satisfy $L_{u,v}$, then applying f to both u and v should result in making $L_{u,v}$ hold. More precisely, let us assume that by some magic we have the ability to simultaneously:

- Apply f to N_u with respect to v;
- Apply f to N_v with respect to u;
- Remove any packets stored in channels $C_{u,v}$ and $C_{v,u}$.

Then the resulting state of the (u, v) subsystem should satisfy $L_{u,v}$. Of course, in a real distributed system such simultaneous actions are clearly impossible. However, we will achieve essentially the same effect by applying a so-called "reset" protocol to the (u, v) subsystem. We will describe a stabilizing local reset protocol for this purpose in the next section.

The first requirement allows nodes u and v to correct the (u, v) subsystem if $L_{u,v}$ does not hold. But other subsystems may be correcting at the same time! Since subsystems overlap, correction of one subsystem may invalidate the correctness of an overlapping subsystem. For example, the (u, v) and (v, w) subsystems overlap at v. If correcting the (u, v) subsystem causes the (v, w) subsystem to be incorrect, then the correction process can "thrash". To prevent thrashing, we add a second requirement. In its simplest form, we might require that correction of the (u, v) subsystem leaves the (v, w) subsystem correct *if* the (v, w) subsystem was correct in the first place.

However, there is a more general definition of a reset function f that turns out to be useful. Recall that we wanted to avoid thrashing that could be caused if correcting a subsystem causes an adjacent subsystem to be incorrect. Informally, let us say that the (u, v) subsystem depends on the (v, w) subsystem if correcting the (v, w) subsystem can invalidate the (u, v) subsystem. If this dependency relation is cyclic, then thrashing can occur. On the other hand if the dependency relation is acyclic then the correction process will eventually stabilize. Such an acyclic dependency relation can be formalized using a partial order $<$ on *unordered* pairs of nodes: informally, the (u, v) subsystem depends on the (v, w) subsystem if $\{v, w\} < \{u, v\}$.

Using this notion of a partial order, we present the formal definition of a local reset function:

Definition 11 *We say f is a local reset function for network automaton $\mathcal{N} = net(G, N)$ with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ and partial order $<$, if for any state s of $\mathcal{N}$ and any edge (u, v) of G:*

- **Correction:** $(f(s|u, v), nil, nil, f(s|v, u)) \in L_{u,v}$.
- **Stability:** *For any neighbor w of v,*

  If $(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v}$ and $\{v, w\} \not< \{u, v\}$ then
  $(s|u, s|(u, v), s|(v, u), f(s|v, w)) \in L_{u,v}$.

Notice that in the special case where all the link subsystems are independent, no edge is "less" than any other edge in the partial order.

Using the definition of a reset function, we can define what it means to be locally correctable.

Definition 12 *A network automaton $\mathcal{N}$ is* locally correctable *to L using link predicate set $\mathcal{L}$, local reset function f, and partial order $<$ if:*

- *$\mathcal{N}$ is locally checkable for L using $\mathcal{L}$.*

- *f is a local reset function for $\mathcal{N}$ with respect to $\mathcal{L}$ and $<$.*

Intuitively, if we have a reset function f with partial order $<$ we can expect the local correction to stabilize in time proportional to the maximum chain length in the partial order. Recall that a chain is a sequence $a_1 < a_2 < a_3 \ldots, < a_n$. Thus the following piece of notation is useful.

Definition 13 *For any partial order $<$, height$(<)$ is the length of the maximum length chain in $<$.*

7 Local Correction Theorem

7.1 Theorem Statement

In the previous section, we set up plausible conditions under which a network automaton can be locally corrected to achieve a property L. We claimed that these conditions could be exploited to yield local correction. In this section we make these claims precise.

Intuitively, the result states that if $\mathcal{N}$ is locally correctable to L using local reset function f and partial order $<$, then we can transform $\mathcal{N}$ into a new *augmented automaton* $\mathcal{N}^+$ such that $\mathcal{N}^+$ satisfies the following property: eventually every behavior of $\mathcal{N}^+$ will "look like" a behavior of $\mathcal{N}$ in which L holds. We use the notation $\mathcal{N}|L$ to denote an automaton identical to $\mathcal{N}$ except that the initial states of $\mathcal{N}|L$ are restricted to lie in set L.

Theorem 2 Local Correction: *Consider any network automaton $\mathcal{N} = Net(G, N)$ that is* locally correctable *to L using link predicate set $\mathcal{L}$, local reset function f, and partial order $<$. Then there exists some $\mathcal{N}^+$ that is a UIOA and a network automaton such that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|L$.*

So far we have ignored time complexity. However, time complexity is of crucial importance to a protocol designer because nobody would be interested in a protocol that took years to stabilize. Thus, [Var92b], uses a version of the timed automata model throughout and uses this model to give a formal definition of stabilization time. However, intuitively, we can measure time complexity as follows. We assume that it takes 1 time unit to send any message on a link and to deliver a free action; we assume that node processing takes time 0. With this assumption, we can define the stabilization time of a behavior to be time it takes before the behavior has a suffix that belongs to the target set. We define the overall stabilization time as the worst case stabilization time across all behaviors.

Using this intuitive definition of stabilization time we can state a more refined version of the Local Correction Theorem. The refinement we add is that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|L$ in time proportional to *height*$(<)$, the height of the partial order.

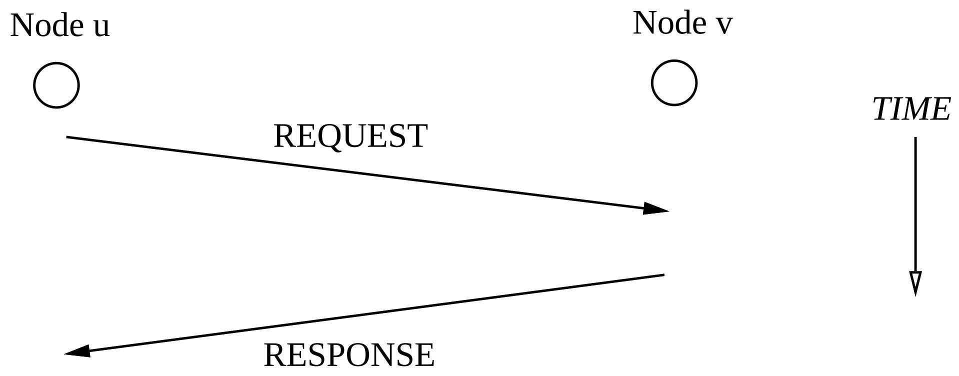

Figure 14: The structure of a single phase of the local snapshot/reset protocol

7.2 Overview of the Transformation Code

To transform $\mathcal{N}$ into $\mathcal{N}^+$ we will add actions and states to $\mathcal{N}$. These actions will be used to send and receive *snapshot packets* (that will be used to do local checking on each link subsystem) and *reset* packets (that will be used to do local correction on each link subsystem). For every link (u, v), the leader $l(u, v)$ initiates the checking and correction of the (u, v) subsystem. The idea is, of course, that the leader of each (u, v) subsystem will periodically do a *local snapshot* to check if the (u, v) subsystem satisfies its predicates; if the leader detects a violation, it tries to make the local predicate true by doing a *local reset* of the (u, v) subsystem.

The structure of our local snapshot protocol is slightly different from the Chandy-Lamport snapshot protocol [CL85] studied earlier in the course. It is possible to show [Var92b] that the Chandy-Lamport scheme cannot be used without modifications over unit storage links. (Prove this: it will help you understand the Chandy-Lamport scheme better.)

Our local snapshot/reset protocol works roughly as follows. Consider a (u, v) subsystem. Assume that $l(u, v) = u$ – i.e., u is the leader on link (u, v). A single snapshot or reset phase has the structure shown in Fig 14.

A single phase of either a snapshot or reset procedure consists of u sending a request that is received by v, followed by v sending a response that is received by u. During a phase, node u sets a flag ($check_u[v]$) to indicate that it is checking/correcting the (u, v) subsystem. While this flag is set, *no packets other than request packets can be sent on link $C_{u,v}$*. Since a phase completes in constant time, this does not delay the data packets by more than a constant factor.

In what follows, we will use the basic state at a node u to mean the part of the state at u "corresponding" to automaton N_u. To do a snapshot, node u sends a snapshot request to v. A snapshot request is identified by a *mode* variable in the request packet that carries a

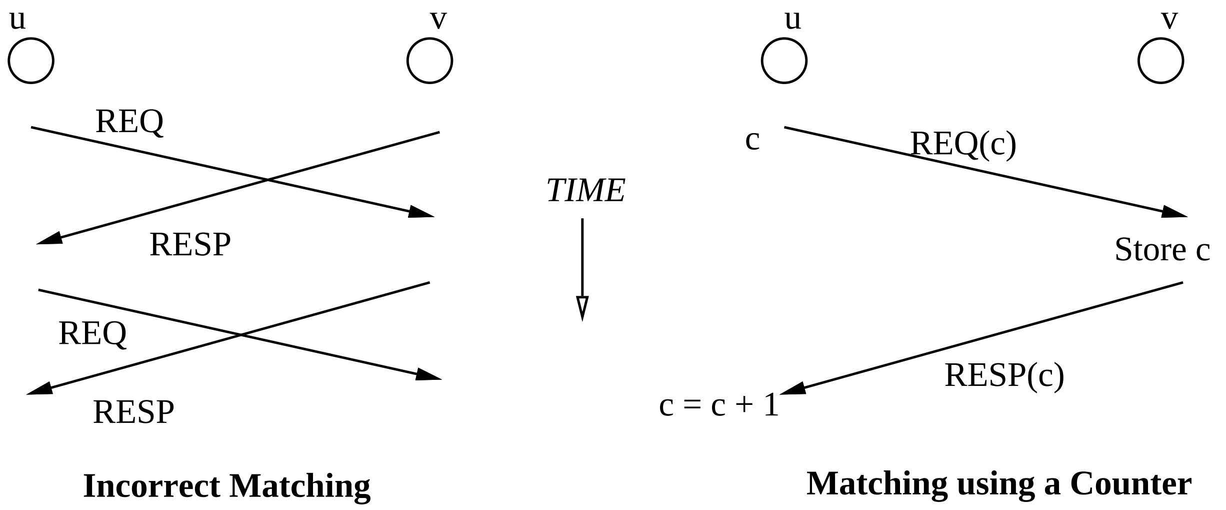

Figure 15: Using counter flushing to ensure that request-response matching will work correctly within a small number of phases.

mode of *snapshot*. If v receives a request with a *mode* of *snapshot*, Node v then records its basic state (say s) and sends s in a response to u.

When u receives the response, it records its basic state (say r). Node u then records the state of the (u, v) subsystem as $x = (r, nil, nil, s)$. If $x \notin L_{u,v}$ (i.e., local property $L_{u,v}$ does not hold) then u initiates a reset.

To do a reset, node u sends a reset request to v. A reset request is identified by a *mode* variable in the request packet that carries a *mode* of *reset*. Recall that f denotes the local reset function. After v receives the request, v changes its basic state to $f(v, s, u)$, where s is the previous value of v's basic state. Node v then sends a response to u. When u receives the response, u changes its basic state to $f(u, r, v)$, where r is the previous value of u's basic state.

Of course, the local snapshot and reset protocol must also be stabilizing. However, the protocol we just described informally may fail if requests and responses are not properly matched. This can happen, for instance, if there are spurious packets in the initial state of $\mathcal{N}^+$. It is easy to see that both the local snapshot and reset procedures will only work correctly if the response from v is sent following the receipt of the request at u. The diagram on the left of Figure 15 shows a scenario in which requests are matched incorrectly to "old" responses that were sent in previous phases. Thus we need each phase to eventually follow the structure shown in the right of Figure 15.

To make the snapshot and reset protocols stabilizing, we use counter flushing. Thus we number all request and response packets. Thus each request and response packet carries a number *count*. Also, the leader u keeps a variable $count_u[v]$ that u uses to number all requests sent to v within a phase. At the end of the phase, u increments $count_u[v]$. Similarly, the responder v keeps a variable $count_v[u]$ in which v stores the number of the last request it has received from u. Node v weeds out duplicates by only accepting requests whose number is not equal $count_u[v]$.

Clearly the *count* values can be arbitrary in the initial state and the first few phases may not work correctly. However, counter flushing guarantees that in constant time a response will be properly matched to the correct request. Because the links are unit storage, there can be at most 3 distinct counter values (one in each link and one at the receiver) stored in the link subsystem. Thus a space of 4 numbers is sufficient to guarantee that eventually (more precisely within 5 phases) requests and responses are correctly matched.

Besides properly matching requests and responses, we must also avoid deadlock when the local snapshot/reset protocol begins in an arbitrary state. To do so, when $check_u[v]$ is *true* (i.e., u is in the middle of a phase), u continuously sends requests. Since v weeds out duplicates this does no harm and also prevents deadlock. Similarly, v continuously sends responses to the last request v has received. Once the responses begin to be properly matched to requests, this does no harm, because u discards such duplicate responses.

8 Intuitive Proof of Local Correction Theorem

A formal proof of the Local Correction theorem can be found in [Var92b]. In this section, we provide the main intuition. We have already described how to transform a locally correctable automaton $\mathcal{N}$ into an augmented automaton $\mathcal{N}^+$ We have to prove that in time proportional to the height of the partial order every behavior of $\mathcal{N}^+$ is a behavior of $\mathcal{N}$ in which all local predicates hold.

We have already described the intuition behind the use of a counter to ensure proper request-response matching. We now describe the intuition behind the local snapshot and reset procedures. The intuition behind the snapshot is very similar to the intuition behind the snapshot procedure studied earlier in the course.

8.1 Intuition Behind Local Snapshots

The diagram on the left of Figure 16 shows why a snapshot works correctly if the response from v is sent following the receipt of the request at u. Let a' and b be the state of nodes u and v respectively just *before* the response is *sent*. Let a and b' be the state of nodes u and v respectively just *after* the response is delivered. This is sketched in Figure 16.

The snhapshot protocol must guarantee that node u does not send any data packets to v during a phase. Also v cannot send another data packet to u from the time the response is

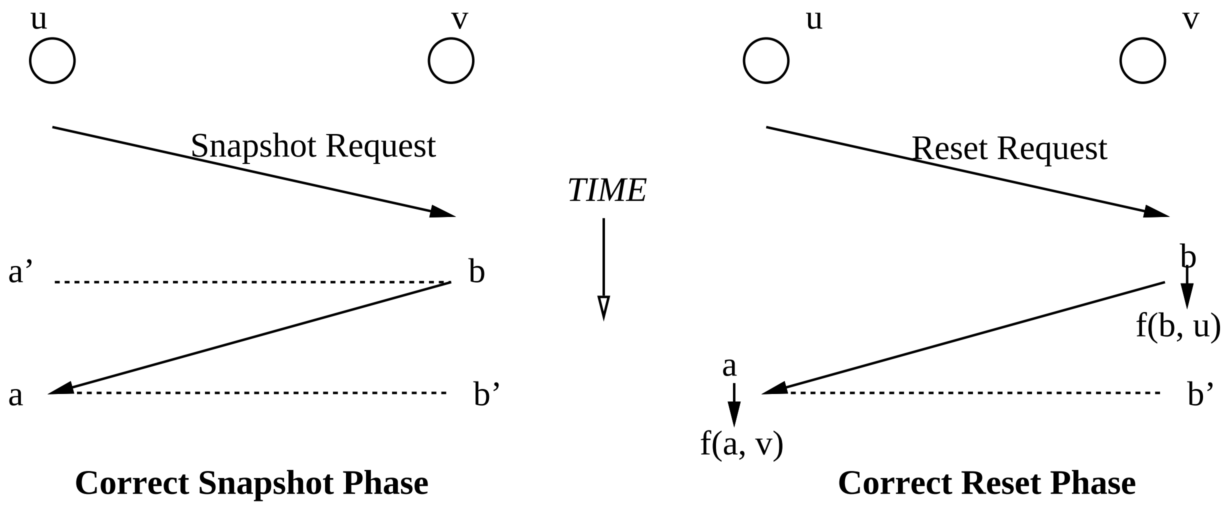

Figure 16: Local Snapshots and Resets work correctly if requests and responses are properly matched.

sent until the response is delivered. This is because the link from v to u is a UDL that will not give a free indication until the response is received. Recall that *nil* denotes the absence of any packet on a link. Thus the state of the (u, v) subsystem just before the response is sent is (a', nil, nil, b). Similarly, the state of the (u, v) subsystem just after the response is delivered is (a, nil, nil, b').

We claim that it is *possible* to construct some other execution of the (u, v) subsystem which starts in state (a', nil, nil, b), has an intermediate state equal to (a, nil, nil, b) and has a final state equal to (a, nil, nil, b'). This is because we could have first applied all the actions that changed the state of node u from a' to a, which would cause the (u, v) subsystem to reach the intermediate state. Next, we could apply all the actions that changed the state of node v from b to b', which will cause the (u, v) subsystem to reach the final state. Note that this construction is only possible because u and v do not send data packets to each other between the time the response is sent and until the time the response is delivered.

Thus the state (a, nil, nil, b) recorded by the snapshot is a *possible* successor of the state of (u, v) subsystem when the response is sent. The recorded state is also a a *possible* predecessor of the state of (u, v) subsystem when the response is delivered. But $L_{u,v}$ is a closed predicate – it remains true once it is true. Thus if $L_{u,v}$ was true just before the response was sent, then the state recorded by the snapshot must also satisfy $L_{u,v}$. Similarly, if $L_{u,v}$ is false just after the response is delivered, then the state recorded by the snapshot *does not* satisfy $L_{u,v}$. Thus the snapshot detection mechanism *will not produce false alarms* if the local predicate holds at the start of the phase. Also the snapshot mechanism *will detect a violation* if the the local predicate does not hold at the end of the phase.

8.2 Intuition Behind Local Resets

The diagram on the right of Figure 16 shows why a local reset works correctly if the response from v is sent following the receipt of the request at u. Let b be the state of node v just *before* the response is *sent*. Let a and b' be the state of nodes u and v respectively just *before* the response is delivered. This is sketched in Figure 16.

The code for an augmented automaton will ensure that just after the response is sent, node v will locally reset its state to $f(b,u)$. Similarly, immediately after it receives the response, node u will locally reset its state to $f(a,v)$. Using similar arguments to the ones used for a snapshot, we can show that there is some execution of the (u,v) subsystem which begins in the state $(f(a,v), nil, nil, f(b,u))$ and ends in the state $(f(a,v), nil, nil, b')$. But the latter state is the state of the (u,v) subsystem immediately after the response is delivered. But we know, from the correction property of a local reset function, that $(f(a,v), nil, nil, f(b,u))$ satisfies $L_{u,v}$. Since $L_{u,v}$ is a closed predicate, we conclude that $L_{u,v}$ holds at the end of the reset phase.

8.3 Intuition Behind Local Correction Theorem

We can now see intuitively why the augmented automaton will ensure that all local predicates hold in time proportional to the height of the partial order. Consider a (u,v) subsystem where $\{u,v\} \not\prec \{w,x\}$ for any pair of neighbors w,x – i.e., $\{u,v\}$ is a minimal element in the partial order. Then, within 5 phases of the (u,v) subsystem the request-response matching will begin to work correctly. If the sixth phase of the (u,v) subsystem is a snapshot phase, then either $L_{u,v}$ will hold at the end of the phase or the snapshot will detect a violation. But in the latter case, the seventh phase will be a reset phase which will cause $L_{u,v}$ to hold at the end of the seventh phase.

But once $L_{u,v}$ remains true, it remains true. This is because $L_{u,v}$ is a closed predicate of the original automaton $\mathcal{N}$ and the only extra actions we have added to $\mathcal{N}^+$ that can affect $L_{u,v}$ are actions to locally reset a node using the reset function f. But by the stability property of a local reset function, any applications of f at u with respect to some neighbor other than v cannot affect $L_{u,v}$. Similarly, any applications of f at v with respect to some neighbor other than u cannot affect $L_{u,v}$. Thus in constant time, the local predicates – corresponding to link subsystems that are minimal elements in the partial order – will become and remain true.

Now suppose that the local predicates for all subsystems with height $\leq i$ hold from some state s_i onward. By similar arguments, we can show that in constant time after s_i, the local predicates for all subsystems with height $i+1$ become and remain true. Once again, the argument depends crucially on the stability property of a local reset function. The intuition is that applications of the local reset function to subsystems with height $\leq i$ do not occur after state s_i. But these are the only actions that can falsify the local predicates for subsystems with height $i+1$. The net result is that all local predicates become and remain true within time proportional to the height of the partial order $<$.

9 A Simple Example of Local Checking: Token Passing

Token protocols that stabilize in time proportional to the height of the tree have been described before by [DolevIM90] in a shared memory setting. However, we believe our protocol is simpler and more transparent. While the mutual exclusion protocol is very simple, its simplicity makes it a good candidate to understand how the general method of local correction can be applied. A more complicated example (network reset protocol) can be found in [Var92a].

Stabilizing spanning tree protocols have been well studied [AKY90],[AG90],[AV91]. Thus we can reduce the problem of token passing on an arbitrary connected network to that of token passing on a tree. Instead of modelling the interface to the network automaton that computes the tree, we will (for simplicity) assume that the network graph is a tree. Thus we are dealing with a network automaton of the form $\mathcal{N} = Net(G, N)$ where G is a tree.

For a tree graph we will refer to the leader function l as *parent* for obvious reasons.

The simplest mutual exclusion algorithm is to pass a token along some tour (e.g., DFS) of the tree; a node can go "critical" when it has the token. But such a protocol is not locally checkable; if there are two tokens at two links, each link subsystem may not locally detect a problem.

Once we see this, it is easy to add a small amount of state to make the token protocol locally checkable. We add a pointer, $pointer_u$ to each node i such that $pointer_u$ points to where the token is in the tree; also if the token is at node u, then $pointer_u = nil$. This is shown in Figure 17.

9.1 Specification of Token Passing Protocol

Our stabilizing token passing protocol is based on this idea and is specified by the automaton $\mathcal{T}$ shown in Figure 18.

We assume that there is a function $nset(u)$ that lists the set of neighboring nodes of node u. (Recall that we allowed node automata to depend on the set of neighboring nodes

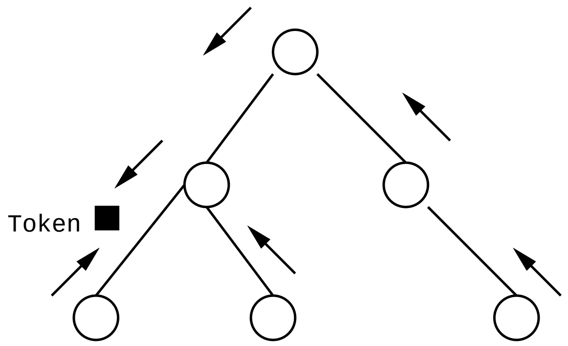

Figure 17: Adding a pointer to each node makes token passing on a tree locally checkable.

and the leader function.) We assume there is some fixed circular ordering on $nset(u)$ and that there is a function $succ_u$ which when given a neighbor v as its argument produces the next neighbor in the ordering. The only packet sent by this automaton is a token packet $token \in P_{data}$. We use the following additional variables:

- $pointer_u \in nset(u) \cup nil$ (*pointer to token*)
- $last_u \in nset(u)$ (*last neighbor u received token from*)
- $queue_u[v]$ (*queue that either contains exactly one *token* packet or is empty.*)
- $free_u[v]$: (*Boolean, true if no packet in transit on link to v *).

Note that we can use domain restriction to ensure that the outbound queue for a link either contains a token or is empty. This can be done by using a single value to encode the queue that is either *token* or *nil*. We use a queue to keep the definition of N_u compatible with the definition of a node automaton given earlier. To make the token passing automaton a node automaton, we also have to pass the token by first enqueueing the token in the outbound queue for a link. Then, a second separate action is required to send the token from the queue to the link. However, by making N_u a node automaton, we can apply the Local Correction Theorem.

Notice the code we have given does not specify start states.

9.2 Adding Local Checking and Correction

To add local checking and correction, we need to define a link predicate set $\mathcal{L}$ for $\mathcal{T}$. And to define $\mathcal{L}$ we need to define a link predicate, $L_{u,v}$, for each edge (u,v) in T. Let $havetoken(u,v)$ be the boolean condition that is true iff there is a token in either $queue_u[v]$, $Q_{u,v}$, $queue_v[u]$, ENQUEUE$_{u,v}$ (*output action to enqueue token in queue for neighbor v*)

Preconditions:

$pointer_u = nil$; (* u has token?*)

$v = succ_u(last_u)$; (* v is the next neighbor after *last* ?*)

Effect:

$pointer_u := v$ (* point towards where token is sent*)

Add *token* to $queue_u[v]$ (*store token in outbound queue*)

SEND$_{u,v}$(*token*) (*output action to send token to neighbor v*)

Preconditions:

$free_u[v] = true$ (* link to v free?*)

token is head of $queue_u[v]$

Effect:

$free_u[v] = false$ (* set to false until link says its free*)

Remove *token* from head of $queue_u[v]$ (*empty outbound queue*)

FREE$_{u,v}$ (*link to v says it is free, input action*)

Effect:

$free_u[v] = true$

RECEIVE$_{v,u}$(*token*) (*input action, token received from neighbor v*)

Effect:

If $pointer_u = v$ then (* token received on expected link?*)

$pointer_u := nil$ (* accept token*)

$last_u := v$ (* update *last**)

All actions are in a separate class with upper bound t_n.

Figure 18: Actions for a node u with respect to a neighbor v in the stabilizing token passing protocol.

or $Q_{v,u}$. (Informally, *havetoken*(u,v) is true iff there is a token stored on any of the links between u and v or on the outbound queues for such links.)

Intuitively, $L_{u,v}$ should describe the legal states of the (u,v) subsystem when $\mathcal{N}$ is behaving properly and there is exactly one token in the system. Consider such a good state and suppose the pointer at u is pointing to v. Then either the token is in transit between u and v OR the pointer at v is pointing away from u.

Definition 14 Local Predicates for Mutual Exclusion: *We define the local predicate $L_{u,v}$ of $\mathcal{T}$ to hold iff all the following conditions hold in the (u,v) subsystem:*

- *Exactly one of $(pointer_u \neq v)$ or $havetoken(u,v)$ or $(pointer_v \neq u)$ is true.*
- *There can be at most one token packet in the combination of $queue_u[v]$, $Q_{u,v}$, $queue_v[u]$ and $Q_{v,u}$.*

Let $\mathcal{L}$ be the link predicate set that consists of $L_{u,v}$ for each edge (u,v) in T. Let L denote $Conj(\mathcal{L})$.

Then we can show that:

Lemma 3 *The network automaton $\mathcal{T}$ is locally checkable for L using link predicate set $\mathcal{L}$.*

Proof: By definition $L = Conj(\mathcal{L})$. We also need to show that each $L_{u,v}$ is a closed predicate. Recall that we say that a state s of $\mathcal{N}$ *satisfies* $L_{u,v}$ iff $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$. Thus we need to show that for any transition $(s, \pi, \tilde{s})$ of $\mathcal{N}$, if s satisfies $L_{u,v}$, then so does $\tilde{s}$. So assume that s satisfies $L_{u,v}$.

Clearly we need only consider actions at nodes u and v since only such actions can affect variables of the (u,v) subsystem. Also we need only consider actions at node u, because the argument for actions at node v is symmetrical.

Suppose π is a $\textsc{Enqueue}_{u,x}$ event. Then in s, $pointer_u = nil$. Thus we can infer that in s, $havetoken(u,v) = false$ and $pointer_v = u$. If $x = v$, then in $\tilde{s}$, $havetoken(u,v) = true$ and $pointer_u = v$ and $pointer_v = u$. Also in $\tilde{s}$ there is no token in $Q_{u,v}$, $queue_v[u]$ and $Q_{v,u}$. On the other hand, if $x \neq v$, then in $\tilde{s}$, $havetoken(u,v) = false$ and $pointer_u \neq v$ and $pointer_v = u$. In either case, $\tilde{s}$ satisfies $L_{u,v}$.

Suppose π is a $\textsc{Send}_{u,x}(token)$ event. If $x \neq v$, then this action does not affect any of the concerned variables. So suppose $x = v$. Then in s, *token* is the head of $queue_u[v]$. Thus we can infer that in s, $pointer_u = x$, $havetoken(u,v) = true$, and $pointer_v = u$. Also in s there is no token in $Q_{u,v}$, $queue_v[u]$ and $Q_{v,u}$. All this action does is to move the token from $queue_u[v]$ to $Q_{u,v}$. Thus $\tilde{s}$ satisfies $L_{u,v}$.

Suppose π is a $\textsc{Free}_{u,x}$ event. Then this action does not change the concerned variables and hence $\tilde{s}$ satisfies $L_{u,v}$.

Empty $queue_u[v]$	(*remove any token stored in queue*)
If u is the parent of v then	
If $pointer_u = v$ then $pointer_u = nil$	(* take away token from child subtree*)
Else	(*v is parent of u*)
$pointer_u = v$	(* point towards parent*)

Figure 19: Code for $f(s|u,v)$, the reset function applied at node u with respect to neighbor v

Suppose π is a $\text{RECEIVE}_{v,u}(token)$ event. Then in s, $Q_{v,u} = token$. Thus we can infer that in s, $pointer_u = v$, $havetoken(u,v) = true$ and $pointer_v = u$. Also in s there is no token in $Q_{u,v}$, $queue_v[u]$ and $queue_u[v]$. Thus in $\tilde{s}$, the only change is that $Q_{v,u} = nil$, $havetoken(u,v) = false$, and $pointer_u = nil$.

The most interesting case is if π is a $\text{RECEIVE}_{x,u}(token)$ event with $x \neq v$. We consider two cases. Suppose in s, $pointer_u \neq x$. Then, since the code will not accept the token in this case, it is easy to see that in $\tilde{s}$, the values of $pointer_u$,$pointer_v$,$Q_{u,v}$, $Q_{v,u}$, $queue_v[u]$ and $queue_u[v]$ are identical to their values in s. Hence $\tilde{s}$ satisfies $L_{u,v}$. Suppose in s, $pointer_u = x$. Then we can infer that in s, $pointer_v = u$ and $havetoken(u,v) = false$. Also in $\tilde{s}$, $pointer_u = nil$ and $pointer_v = u$ and $havetoken(u,v) = false$. Thus $\tilde{s}$ satisfies $L_{u,v}$. $\square$

The next thing we have to do is to specify a local reset function f to correct each (u,v) subsystem. Our idea is very simple. Let us define a partial order on pairs of neighbors in the tree T such that "any edge e is less than any edge below e in T". More precisely, we let $\{u,v\} \prec \{v,w\}$ iff u is the parent of v and v is the parent of w. Also, $\{u,v\} \not\prec \{w,x\}$ if the two pairs do not have a node in common. We let $<$ be the transitive closure of $\prec$. Thus, the partial order directly reflects the structure of the tree T.

To allow f to be a reset function using $<$ we must ensure that applying f to the state of v with respect to child w does not affect the stability of $L_{u,v}$. This can be achieved by the following reset function f described in Figure 19.

Lemma 4 *The function f defined in Fig. 19 is a local reset function for network automaton $\mathcal{T}$ with respect to link predicate set $\mathcal{L}$ and partial order $<$.*

Proof: Consider any edge (u,v) and suppose that u is the parent of v; the reverse case is symmetrical. We check the two conditions in Definition 11.

- **Correction:** We need to show that $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$. Consider $f(s|u,v)$. From the code in Figure 19 we see that in this node state, $pointer_u \neq v$ and $queue_u[v]$ is empty. Consider $f(s|v,u)$. From the code in Figure 19 we see that in this node state, $pointer_v = u$ and $queue_v[u]$ is empty. Thus $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$.

- **Stability:** We only need to check the stability condition for links less than (u,v) in the partial order. Since v is the child of u it suffices to show the following: for any neighbor w of v, if $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$ then $(s|u, s|(u,v), s|(v,u), f(s|v,w)) \in L_{u,v}$. But if we change the state of node v from $s|v$ to $f(s|v,w)$, $queue_v[u]$ remains unchanged. Next, if $pointer_v \neq u$ in $s|v$, then $pointer_v \neq u$ in $f(s|v,w)$. Similarly, if $pointer_v = u$ in $s|v$, then $pointer_v = u$ in $f(s|v,w)$. Taken together, these facts imply the stability condition.

□

Let $height(T)$ denote the height of tree T which in turn is the length of the longest path from the root to a leaf. Clearly $height(f) = height(T)$.

The next lemma that follows directly from the previous lemmas and the definitions:

Lemma 5 *The network automaton $\mathcal{T}$ is locally correctable to L using link predicate set $\mathcal{L}$ and the reset function f defined in Fig. 19.*

By applying the Local Correction theorem (Theorem 2) (and especially the transformation underlying its proof) it is easy to see that $\mathcal{T}$ can be transformed into another automaton $\mathcal{T}^+$ that does local checking and correction. Also $\mathcal{T}^+$ stablizes to a version of $\mathcal{T}$ in which all local predicates hold in time proportional to the height of the tree.

Finally, we show that if all local predicates hold, there is exactly one token and token passing works correctly.

To make a verbal argument, we introduce some intuitive terminology. Let us say that a token is in transit between nodes u and v if $havetoken(u,v) = true$. We say that a token is at node u if $pointer_u = nil$.

We first show that for any $s \in L$, there is at least one token in s. An intuitive explanation for this is as follows. We will define a search procedure to find a token in state s. Start at any node u in the tree. If $pointer_u = nil$ then there is a token at u. If $pointer_u = v$ then from the local predicates, either there is a token in transit between u and v or $pointer_v \neq u$. If $pointer_v \neq u$ we continue the search procedure recursively at node v. Since we never backtrack, the search procedure cannot continue indefinitely without encountering some leaf node w such that $pointer_w \neq x$, where x is the parent of w. But if w is a leaf node and $pointer_w \neq x$ then, $pointer_w = nil$ and hence the token is at w.

Thus we know that there is at least one token in state s. Suppose this token is at node u. Then by induction on the length of the path between u and any node $w \neq u$, it is easy to see that $pointer_w \neq nil$. Similarly for any edge (w,x), by induction on the length of the

RECEIVE$_{v,u}$(*token*) (*token is received from neighbor v*)

Effect:

$pointer_u := nil$ (* accept token*)

$last_u := v$ (* update *last**)

All external actions are in a separate class with upper bound t_n.

Figure 20: Modified Code for a node u in a token passing protocol. The remaining code is identical to the code in Figure 18

path between u and (w, x) we can show that the token is not in transit between w and x. A similar argument shows that if in s the token is in transit between u and v, then the token is not at any node x, nor is it in transit on any other edge. (w, x). Hence there is exactly one token in state s.

Once we know that there is exactly one token in any state s it is quite easy to prove that every node receives the token periodically.

9.3 Removing Unexpected Packet Transitions

Consider the following modification of the token passing protocol described in Figure 18. The modification is shown in Figure 20. The only difference is that the routine to receive a token at a node u from node v has been changed. The only change is that we no longer check whether $pointer_u = v$ before accepting the token. It is not hard to show that if we remove this check, the local predicates $L_{u,v}$ from Definition 14 are not closed predicate. In other words, omitting this simple check is sufficient to make the token passing automaton no longer locally checkable. Yet, this check is not required when the tree automaton is behaving correctly (because the token will only be received from the link being pointed to).

The point is that sometimes we have to add extra defensive checks to a protocol in order to make sure that its local predicates remain true once they are true. These checks, which we call checks for unexpected packet transitions, are not needed when the protocol is behaving correctly.

We would like to abstract this process. First, note that the extra check added essentially amounts to the following heuristic: *if we receive an "unexpected" packet p at node u from node v, then we ignore packet p.* Notice that a token received from v when $pointer_u \neq v$ (see Figure 21) is certainly "unexpected". We will formalize this intuitive notion of an "unexpected" packet below. However, for the present it is important to intuitively understand

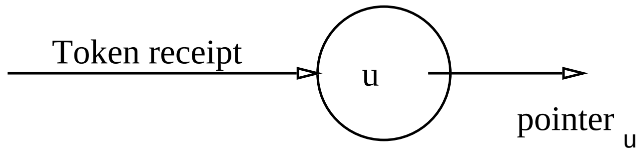

Figure 21: Receiving a token on a link that is not being pointed to is an unexpected packet transition. In general, an unexpected packet transition is a packet reception that could never have occurred if the receiving link subsystem was in a good state.

why such checks for unexpected packets are useful. Consider three nodes u, v, w and some state s such that s satisfies $L_{u,v}$ but not $L_{u,w}$ – i.e., the link from u to v, is working correctly but not the link from w to v. Then it is quite possible for u to receive an "unexpected" packet from node w in state s. If u accepts this packet and processes it, this may result in making $L_{u,v}$ false. By adding checks for such packets in $\mathcal{N}$, we can ensure that $\tilde{s}$ satisfies $L_{u,v}$. Also note that these checks do not affect the "correct" executions of $\mathcal{N}|L$.

We formalize this notion of an "unexpected packet" (see Figure 21) as follows:

Definition 15 *Consider a network automaton $\mathcal{N}$ with link predicate set $\mathcal{L} = \{L_{u,v}\}$ and some pair of neighbors u, v in $\mathcal{N}$. We say that a transition $(s, \pi, \tilde{s})$ is an unexpected packet transition at u with respect to v if:*

π is a $\text{RECEIVE}_{v,u}(p)$ event and there is no a, b such that $(s|u, a, p, b) \in L_{u,v}$.

In other words, a packet p received by node u from node v in state s is unexpected if there is no "good" state of the (u, v) subsystem that is compatible with the receipt of this packet.

We have found that adding checks for such unexpected packets is a a powerful heuristic in order to ensure that local predicates remain true once they are true. We also have identified some precise sufficient conditions that partly explain why this heuristic is useful. Essentially these checks are a simple form of defensive programming.

10 Other Theorems on Local Checking

There are three other major theorems in [Var92b] that revolve around local checking. For the formal statements, recall that a UIOA is an automaton which can start in any state

Dijkstra's second example suggests that any locally checkable protocol on a tree can be stabilized. In the message passing model, we can call this the *Tree Correction Theorem*. This theorem states that any locally checkable protocol on a tree can be efficiently stabilized in time proportional to the height of the tree. In other words, if the underlying topology is a tree we can dispense with the need for local correctability.

Theorem 6 Tree Correction: *Consider any automaton $\mathcal{T} = Net(T, N)$ such that T is a rooted tree. Suppose also that $\mathcal{T}$ is* locally checkable *for L using link predicate set $\mathcal{L}$. Then there exists some network automaton $\mathcal{N}^+$ that is a UIOA such that $\mathcal{N}^+$ stabilizes to $\mathcal{N}|L$ in time proportional to the height of the tree.*

The next result result is the *Global Correction* theorem. This theorem states that any locally checkable protocol can be stabilized in time proportional to the number of network nodes. The Global Correction theorem shows that we can dispense with the need for local correctability and the need for the underlying topology to be a tree as long as we are willing to pay a higher price in stabilization time. The height of the underlying partial order in the local correction method and the height of the tree in in the tree correction method are typically smaller than the number of network nodes. Thus it pays to use the local correction or tree correction wherever possible. Formally:

Theorem 7 Global Correction: *Consider any network automaton $\mathcal{N} = Net(G, N)$ that is* locally checkable *for some predicate L using link predicate set $\mathcal{L}$. Then there exists some network automaton $\mathcal{N}^+$ that is a UIOA and such that$\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}|L$ in time proportional to the number of network nodes.*

The transformation underlying the Global Correction Theorem is very similar to that underlying the Local Correction Theorem. Once again we add actions to every link to do a local snapshot of every link; however, if a violation is detected, instead of doing a local reset we do a global reset of the network. The increase in stabilization time is a direct result of using "global" instead of "local" correction.

Compare the statement of the local, tree, and global correction theorems. All three of these theorems formalize *practical* techniques for designing stabilizing protocols. We will talk about implementation issues at the end of this lecture.

Finally, [Var92b] describes two compilers that can convert any synchronous protocol π into a self-stabilizing asynchronous version of π. The main compiler, the Resynchronizer, works by first applying the synchronizer protocol of [Awe85] to create an asynchronous version of π. Next we use global correction to make the resulting protocol stabilizing. This can be done by using a stabilizing reset protocol to periodically restart an asynchronous version of protocol π. A more complicated version of the Resynchronizer protocol was originally reported by us in [AV91].

11 Example of Global Correction: A Simple Spanning Tree Protocol

We provide a simple example of the use of the Global Correction Theorem by describing a protocol which we know is locally checkable but appears not be locally correctable.

We begin by describing earlier stabilizing spanning tree protocols and their disadvantages. Then we describe a locally checkable spanning tree protocol. This protocol will be used to construct a stabilizing spanning tree protocol

11.1 Previous Work on Spanning Tree Protocols

The basic idea in virtually all spanning tree algorithms is that nodes report the smallest node ID seen so far (and the shortest distance to this smallest ID node) to their neighbors. Each node then picks as its parent the neighbor that knows of the smallest ID. If more than one neighbor reports the smallest ID, the node picks from among these the neighbor that reports the smallest distance. If two neighbors report the same distance and root ID, then an arbitrary tie-breaker is used to select the parent. A node sets its estimate of the root ID to equal its parent's root ID, and updates its distance from the root to be one plus the parent's distance. Because of the way the distance from the root is calculated, this approach is basically an adaptation of the Bellman-Ford Algorithm.

However, in a dynamic network (and in a stabilizing setting), this approach encounters an obstacle known as "ghost roots". This phenomena occurs whenever the root crashes: its ID, which was the smallest in the system, is still the smallest from the point of view of nodes that do not know of its crash. Even in a static network, the same effect can be caused by initial errors that introduce a root ID lower than the ID of any network node. This ID can potentially remain forever in the system! Even if the nodes maintain a counter to reflect their distance from the alleged root, this counter will simply grow unboundedly. Since the ghost root phenomenon is not a rare event, several ways to overcome this difficulty have been suggested.

One solution, known as the "hop counter" approach, is to have some pre-determined bound on the diameter of the network at each node, and to discard the old root whenever the associated counter reaches some limit (see, e.g.,[AG90]). Unfortunately, this pre-determined bound must be quite high, and hence, the stabilization time of such counting up schemes is poor in practice [CRKG89a, CRKG89b, Gar89, RF89, Awe90].

Another widely used stabilizing Spanning tree protocol is the IEEE 802.1 bridge routing protocol which is based on the design in [Per85]. This solution uses an approach that we call timer flushing. The basic idea [Per85] is that each node "times out" information received from its neighbors unless the information is refreshed periodically. Any node that thinks it is the root is responsible for periodically sending updates to all its neighbors in order to refresh their state information. Any other node X sends estimates to its neighbors only after receiving a message from the node X thinks is its parent. The upshot is that if there is an old root in the system, the information about this old root will eventually "time out", after which the system will stabilize.

However, timer flushing suffers from several drawbacks. First, the node clocks (by which packet lifetimes are enforced) can have different rates. Second, the message delivery time over links typically has large variance; since the topology of the network is not known in advance, the variance of the message delivery time across the whole network is even higher. A conservative timeout bound must take into account high message latencies and the worst-case topology, even though the latencies of an actual execution may be considerably (perhaps even an order of magnitude) smaller. This leads to an order of magnitude slowdown of the stabilization process. Lastly, the parameters of such "global" timeout protocols often have complicated dependencies. Tuning these parameters can be quite difficult.

Th stabilizing spanning tree protocol we will describe is extremely similar to the two schemes we have described above. However, it uses a different mechanism for detecting and recovering from states with ghost roots that speeds up the stabilization time of the resulting protocol.

The detection mechanism is based on the following observation. Consider an execution of the simple spanning tree protocol that starts with a state in which all nodes are correctly initialized and there are no messages in transit on the links. Now focus on some node. Throughout the execution the node maintains a current estimate of the root ID, and another estimate for its distance from this alleged root. It can be shown that in the course of legal executions, the node's estimate of the root ID never goes up; also while such a root estimate is fixed, the distance estimate never goes up. This property can be cast in the form of a local predicate for each link. If the predicate holds, then the algorithm will produce a spanning tree. This immediately suggests the stabilizing algorithm: whenever the predicate is violated, the node that detects the violation makes a reset request. In the execution that follows the last reset signal, all the information will be correct.

Rather than directly describe the final spanning tree protocol, we will derive it in the following way. In the next subsection, we will describe a locally checkable (but non-stabilizing) spanning tree protocol on which the final spanning tree protocol is based. Of course, the Global Correction Theorem is applicable to this protocol. However, we will show in the following section that the spanning tree protocol is also *one-way checkable* which allows a simpler form of checking.

11.2 Code for Locally Checkable Spanning Tree Protocol

We now describe the code for our locally checkable (but non-stabilizing!) spanning tree protocol. After all local predicates hold, this protocol computes a spanning tree in time proportional to the diameter of the network.

Before we delve into the code of the protocol, we define what it means for a spanning tree protocol to be stabilizing. We assume that the network topology is described by some graph G. We say the protocol computes a spanning tree of G in time t if:

- Each node has a parent variable and the graph and the parent variables at nodes do not change after time t.
- The graph induced by the values of the parent variables is a spanning tree of G.

Our locally checkable spanning tree protocol is described by the automaton T_u shown in in Figure 22. The protocol is based on the simple idea alluded to earlier. Each node keeps track of the smallest node ID seen so far and the shortest distance to this smallest ID node. Each node uses this information to compute its parent in the tree.

Thus each node u maintains a parent pointer $parent_u$, current estimate of the root's identity r_u, and a current distance estimate d_u. We denote by (r_u, d_u) the ordered pair at node u. We will use lexicographic ordering for 2-tuples and 3-tuples. For example, $(r_v, d_v) < (r_u, d_u)$ means that either $r_v < r_u$, or that $r_v = r_u$ and $d_v < d_u$. Similarly, $(r_v, d_v, parent_v) < (r_u, d_u, parent_u)$ means that that either $(r_v, d_v) < (r_u, d_u)$, or that $(r_v, d_v) = (r_u, d_u)$ and $parent_v < parent_u$. Each node also maintains a copy (possibly outdated) of the root and distance estimates of each its neighbors. Thus the estimates of neighbor v are stored at u in the variables $r_u[v]$ and $d_u[v]$.

For compactness, we let both arrays have an entry for the node itself, in which the default values are "hardwired": thus $r_u[u] = u$, and $d_u[u] = -1$. Now a node always chooses its own estimate based on the minimum of the estimates of all its neighbors and its own default estimate. Node u chooses its own estimate of the root from this minimum estimate; u also chooses its own estimate of the distance as one plus the distance from the minimum estimate. Thus if u itself is the smallest root, the distance u chooses for itself will be $-1+1 = 0$, which is as it should be. Thus $d_u[u]$ is set to -1 simply as a sentinel value to avoid an extra case.

Every node u periodically sends its own estimate of root and distance to all its neighbors using an "Announce" packet. To make the T_u a node automaton, we have followed the convention of first enqueuing the "Announce" packet on an outbound queue for the link called $queue_u[v]$; then the packet is sent out from the queue when the link is free. The Announce packet is encoded as a tuple $(Announce, r, d)$. When a node u receives an Announce packet from v, u first checks whether the estimate in the packet is greater than the previous estimate stored from v and the distance in the estimate is not already at the maximum possible value. If this is not the case, u stores the latest estimate it has received from v; u also updates its own estimate as the minimum of all neighbor estimates.

Let T_u be the automaton shown in in Figure 22. We assume that for any u, v, all actions of the form $\text{SEND}_{u,v}(*)$ are in a separate class, and any $\text{ENQUEUE}_{u,v}$ action is in a separate class.

Let $\mathcal{T}$ be the composition of the automata node T_u for every node in G with a UDL for every edge (u, v) in G. In the next subsection we show that $\mathcal{T}$ is locally checkable for a predicate L that we define. In the following subsection, we show that $\mathcal{T}|L$ stabilizes to the stable tree behaviors in time proportional to the diameter of graph G. This sets the stage for applying the global correction theorem to $\mathcal{T}$.

11.3 Local Predicates for $\mathcal{T}$

We state a set of local predicates for the spanning tree protocol. Let us denote by $O_{u,v}$ the intersection of the local predicates shown in Figure 23 for any edge (u, v). Note that $O_{u,v}$ consists of two predicates, $O1(u, v)$ and $O2(u, v)$. $O1(u, v)$ says that the sentinel values that node u uses (as a neighbor estimate from itself) are correct. It also says that node u's estimate is the minimum of all its neighbors.

$O2(u, v)$ shows that the sequence of estimates sent from u to a neighbor v are non-increasing. Suppose there is an $(Announce, r, d)$ packet in transit from u to v, then the estimates that v already has stored from u can be no less than the estimates in the packet (i.e., $(r_v[u], d_v[u]) \geq (r, d)$). Also the estimates in the packet can be no less than the estimates at u (i.e., $(r, d) \geq (r_u, d_u)$). If there is no $(Announce, *, *)$ packet in transit from u to v, then the estimates that v already has stored from u can be no less than than the current estimates at u (i.e., $(r_v[u], d_v[u]) \geq (r_u, d_u)$). Recall that a Unit Capacity Link only allows one outstanding packet on each link, just as in a UDL.

Definition 16 *For any edge (u, v), we let $L_{u,v}$ be the intersection of $O_{u,v}$ and $O_{v,u}$. Let $\mathcal{L}$ be the link predicate set containing $L_{u,v}$ for every (u, v) in G and let $L = Conj(\mathcal{L})$.*

It is easy to show that each $L_{u,v}$ is a closed local predicate.

Lemma 8 *For a leader edge (u, v) and transition (s, a, s') of $\mathcal{T}$, if s satisfies $L_{u,v}$, then s' satisfies $L_{u,v}$.*

Proof: First, $O1(u, v)$ is clearly closed because the code never changes the value of $r_u[u]$ and $d_u[u]$. Also, if the code changes $(r_u, d_u, parent_u)$, it always sets this 3-tuple equal to the minimum of $(r_u[v], d_u[v], parent_u[v])$ for all v in $nset_u$ as required by $O1(u, v)$.

<u>State</u>

$nset_u$	set containing all neighbors of u and u itself
$parent_u$	parent pointer, in $nset_u$
d_u:	distance estimate, integer in the range $0..n'$, n' is an upper bound on number of nodes.
r_u:	root estimate, all root estimates are in the range of node identifiers
$r_u[v]$:	estimate of r_v for each v in $nset_u$ except $r_u[u] = u$
$d_u[v]$:	estimate of d_v for each v in $nset_u$ except $d_u[u] = -1$
$free_u[v]$	boolean, true if link to v is free for each v in $nset_u$ except u itself.
$queue_u[v]$	a queue containing at most one $(Announce, *)$ packet to be sent to v.

<u>Actions</u>

$\text{FREE}_{u,v}$ (*input action to receive free notification from link to neighbor v*)
    Effects: $free_u[v] = true$

$\text{SEND}_{u,v}(p)$ (*output action to send estimate to neighbor v*)
    Preconditions: p is the head of $queue_u[v]$ and $free_u[v] = true$
    Effects: $free_u[v] = false$; Remove p from $queue_u[v]$

$\text{ENQUEUE}_{u,v}$ (*output action to enqueue estimate to neighbor v*)
    Preconditions: $queue_u[v]$ is empty
    Effects: Add $(Announce, r_u, d_u)$ to $queue_u[v]$

$\text{RECEIVE}_{v,u}(Announce, r, d)$ (*input action to receive estimate from neighbor v*)
    Effects:
        If $(r, d) \leq (r_u[v], d_u[v])$ then (*received estimate is no greater than previous estimate*)
            If $(d < n')$ (*received estimate not at max value *)
                $(r_u[v], d_u[v]) := (r, d)$ (*update estimate from v*)
                $(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in nset_u\}$

Figure 22: Spanning tree protocol. Code for a node u.

O1(u,v):

$r_u[u] = u$

$d_u[u] = -1$

$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in nset_u\}$

O2(u,v):

If there is an $(Announce, r, d)$ packet in transit from u to v then

$(r_v[u], d_v[u]) \geq (r, d) \geq (r_u, d_u)$

Else (*there is no Announce packet in transit*)

$(r_v[u], d_v[u]) \geq (r_u, d_u)$

Figure 23: Spanning Tree Protocol: Local Predicates for edge (u, v).

Next, $O2(u, v)$ is closed because of a simple observation: *the* (r, d) *estimates sent in* $(Announce, r, d)$ *packets by node* u *are always non-increasing.* This follows because node u changes its own estimate after receiving an *Announce* packet from some neighbor w. But (see code), u will not accept the packet from w unless the new estimate from w is no greater than the previous value that u has stored from w and if the distance from w is not already at the maximum value n'. But if the new estimate from w is no greater than the previous value that u has stored from w, then the new estimate that u calculates will be no greater than before. But if $O2(u, v)$ is true initially and all estimates sent by u are non-increasing, then it is easy to see that $O2(u, v)$ remains true. Note that if u does not accept a packet from neighbor v (e.g. because the distance estimate in the packet is equal to n') this does not affect $O2(v, u)$

Finally, since the above argument can be applied symmetrically for the link (v, u), it follows that $L_{u,v}$ is closed. □

Note that the check in the code that prevents v from accepting an estimate larger than a previously stored estimate is really an application of the heuristic of removing unexpected packet transitions (see earlier). Clearly if u receives an estimate from w larger than the previous estimate stored from w, then $O2(w, u)$ is not true in the state before the estimate was received. Thus this is an unexpected packet transition; by removing such a transition, we ensure that $L_{u,v}$ is a closed predicate.

The following theorem is immediate from the last lemma.

Theorem 9 *The network automaton* $\mathcal{T}$ *can be locally checked for* L *using* $\mathcal{L}$*.*

11.4 Fast Computation of Spanning Tree after all Link Predicates Hold

We now show that $\mathcal{T}|L$ computes a spanning tree in time proportional to the diameter of G. Recall that $\mathcal{T}|L$ is the spanning tree automaton described in Figure 22 such that all local predicates hold in the initial state.

The basic idea is to show first that if all local predicates hold then there can be no ghost roots; next, we show that if the spanning tree protocol begins in a state that does not contain a ghost root, then the protocol quickly converges to a stable spanning tree.

We assume that each node has a unique ID that is modelled by the name of the node in the topology graph. Thus the unique ID of node u is u. Our spanning tree protocol depends heavily on the fact that each node has a unique ID that cannot be corrupted. Thus we are assuming that the node ID is part of the code at a node.

We first define what it means to have a ghost root in a state of $\mathcal{T}$. Informally, consider the set of all the IDs present (in all the root estimates at nodes and in all the *Announce* packets in the links) in state s. If the minimal ID of all nodes in the network is not the the minimal ID in this set, then we have a ghost root. Formally:

Definition 17 *Consider a state s of $\mathcal{T}$. We say that s has a ghost root r if $r < u$ for all nodes u in G and either:*

- *There is an $(Announce, r, *)$ packet in transit from u to v in s for some u, v, OR*
- *$r_u[v] = r$ in s for some u, v, OR*
- *$r_u = r$ in s for some u.*

The only way we can have ghost roots in any execution is because of bad data in links and nodes when the protocol starts up.

Lemma 10 *Consider any execution α of $\mathcal{T}$. Suppose there is a state s_i in α such that there is no ghost root in s_i. Then then there is no ghost root in the state following s_i in α.*

Proof: From the code it is easy to see that the only way a new value is added to set of existing roots is if a node u adds its own ID to the set (e.g., by changing its root estimate to the default). Thus ghost roots cannot be created after state s_i. $\square$

Next, we show that any state in which all local predicates hold cannot contain a ghost root.

Lemma 11 *There is no ghost root in any state of $\mathcal{T}|L$.*

Proof: Suppose we did have a ghost root in some state s of $\mathcal{T}|L$. Then there must be a ghost root r with the lowest ID among ghost roots in s. If it is in an *Announce* packet in transit from say v to u, then we know from $O2(v,u)$ that $r_v \le r$ and hence $r_v = r$. On the other hand, if there is some w such that $r_w[x] = r$ or $r_w = r$, then we know from $O1(w,*)$ that $r_w \le r$ and hence $r_w = r$. Thus, we have some node, say w, such that $r_w = r$.

By $O1(u,v)$ if $parent_w \ne w$ we can continue inductively following parent pointers. Each time we move from say w to $x = parent_w$, we know from $O1(w,*)$ that $(r_w, d_w) = (r_w[x], d_w[x]+1)$. But since $r_w = r$, $(r_w[x], d_w[x]) < (r, d_w)$. But we know from $O2(x,w)$ that that $(r_x, d_x) \le (r_w[x], d_w[x])$. Thus we conclude that $(r_x, d_x) < (r, d_w)$.

But the distance estimates are non-negative and the root estimates in this chain are all equal to r since r is the lowest ID ghost root. Hence the process of following *parent* pointers must terminate at some node say z such that $parent_z = z$ and $r_z = r$. Thus by $O1$ applied to z, we know that $r = z$. But that contradicts the fact that r is a ghost root. $\square$

The last important fact to observe is that if there are no ghost roots and all local predicates hold, then the protocol converges in time proportional to the diameter of the network. As usual, we will say that $t = O(d)$ to mean $t \le cd$ where c is a constant. We denote the diameter of G by D.

Lemma 12 *Consider an execution α of $\mathcal{T}|L$. Within time $t = O(D)$, $\mathcal{T}|L$ has calculated a spanning tree of the network and the values of the parent variables at each node will remain unchanged thereafter.*

Proof: We know from from Lemma 11 that there is no ghost root in the initial state of α and hence by Lemma 10 there is no ghost root in any state of α. From the fact that $O2(u,v)$ is closed, we know that $O2(u,v)$ holds for all links (u,v) in all states of α. Let q be the node with the smallest ID in graph G. We prove the lemma using induction on the distance d of a node u from q.

Inductive Hypothesis: There exists some constant c such that within $c \cdot d$ time after the start of α, there is some state in which $(r_u, d_u, parent_u) = (q, d, v)$, where v is some neighbor of u at distance $d-1$ from q. Also u will not change r_u,d_u or $parent_u$ from this state onwards in α.

The inductive hypothesis implies the lemma because it implies that within time proportional to the diameter, every node has a parent pointer that points to a node that is one hop closer to the root q. It also implies that the parent pointers do not change after this point.

Basis, $d = 0$: Node q is the only node at distance 0 from itself. In all states s_i of α, $(r_q, d_q, p_q) = (q, 0, q)$, or there would be a ghost root in state s_i which we have already ruled out.

Inductive Step: Suppose it is true for all nodes at a distance d from s and we wish to show that it is true for a node v at a distance $d+1$ from s. Thus there is some state s_i such that all nodes u within distance d from q have $(r_u, d_u, parent_u) = (q, d, *)$. Also, s_i occurs within $c \cdot d$ time after the start of α. We first claim that in all states of α, $(r_v, d_v) \geq (q, d+1)$. Suppose this were not true in some state s_k of α. Then by following parent pointers as we did in the proof of Lemma 11 and by using predicates O1 and O2 iteratively, we can show that we must end in a ghost root. Since we have ruled out ghost roots, $(r_v, d_v) \geq (q, d+1)$ in all states of α.

Next, v must have some neighbor at a distance d from q. Let u be the neighbor with the lowest ID among the neighbors of v at distance d from q. Thus by the properties of a UDL, within constant time after state s_i, an $(Announce, q, d)$ packet from u will arrive at v which will cause $(r_v, d_v) = (q, d+1)$. (Note that this packet will be accepted at v because in the previous state $(r_v, d_v) \geq (q, d+1)$ as we have just shown and $d+1 < n'$.) Also since u is the lowest ID neighbor at distance d from q, v will set $parent_u = u$ and this will remain unchanged for the rest of the execution. □

We can now apply the Global Correction Theorem to obtain a stabilizing version of $\mathcal{T}$. However, there is an even simpler transformation because $\mathcal{T}$ is *one-way checkable.*

11.5 One Way Predicates and Local Checking

We have claimed that every locally checkable protocol could be stabilized using a global reset protocol. We suggested that this could be achieved by periodically doing a local snapshot of each local subsystem and making a reset request if a violation is detected.

However, in this section we will show that if a locally checkable protocol P is also what we call *one-way checkable*, then we can locally check P using a simpler and faster method than doing a local snapshot. In this method, each node periodically sends its entire state to each of its neighbors. We can call this *local checking by periodic sending of state* or simply periodic sending. The question that remains is: when is periodic sending adequate to detect local violations in lieu of a local snapshot? The answer is that periodic checking is sufficient if each local predicate is what we call a *separable* local predicate.

Intuitively, a separable local predicate can be separated into two one-way predicates, one for each direction of a link subsystem. Intuitively a *one-way predicate* for link (u, v) is a predicate that only involves the state of u, the state of the link (u, v), and the state of node v. *Note that it does not depend on the the state of the link* (v, u). Formally:

Definition 18 *Consider any network automaton $\mathcal{N}$ with graph G. Let (u, v) be any edge in G. We say that $O_{u,v}$ is a one-way predicate for edge (u, v) of $\mathcal{N}$ if:*

- *$O_{u,v}$ is a local predicate of $\mathcal{N}$ for edge (u, v).*

- *For any two states s and s', suppose s satisfies $O_{u,v}$ and $s|u = s'|u$ and $s|v = s'|v$ and $s|(u,v) = s'|(u,v)$. Then s' satisfies $O_{u,v}$.*

For example, consider the second predicate of the spanning tree protocol listed in 23. Recall that it essentially says that the estimate values in transit from v to u are always non-increasing. Clearly this is a one-way predicate. It is not hard to see that the first predicate (which involves only the state of one node) is also a one-way predicate.

Definition 19 *Consider any network automaton $\mathcal{N}$ with graph G. Let (u,v) be any edge in G. We say that $L_{u,v}$ is a separable local predicate for edge (u,v) of of $\mathcal{N}$ if there exist two one-way predicates $O_{u,v}$ and $O_{v,u}$ (one for edge (u,v) and edge (v,u) respectively) such that:*

Any state s of $\mathcal{N}$ satisfies $L_{u,v}$ iff s satisfies both $O_{u,v}$ and $O_{v,u}$

Once again it is easy to see that the spanning tree protocol has a link predicate set that consists of separable predicates. In this case, we say that the spanning tree protocol is *one-way checkable.*

Definition 20 *A network automaton $\mathcal{N}$ is one-way checkable for predicate L using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{N}$ is locally checkable for predicate L using link predicate set $\mathcal{L}$.*
- *Each $L_{u,v} \in \mathcal{L}$ is a separable local predicate.*

We now claim the following fact informally. *Any one-way checkable automaton can be locally checked by periodic sending.* We simply add actions to send the state of a node periodically to each of its neighbors and actions to receive such packets.

Suppose that in state s, node u receives such a periodic packet from node v containing the value x. Then u checks whether $(x, nil, nil, s|u) \in O_{v,u}$; if this is not true u detects a local violation.

Notice that local checking by periodic sending is not just simpler than doing a local snapshot but is also *faster*. Thus *for one-way checkable protocols it always pays to use periodic sending for local checking.*

12 Implementing Local Checking in Real Networks: Timer Flushing

Earlier we made the snapshot/reset protocol stabilizing by numbering snapshot requests and responses. We also relied on the fact that each link was a UDL. In practice, however, there is an even simpler way of making the snapshot/reset protocol stabilizing. This can be done using timers.

Suppose there is a known bound on the length of time a packet can be stored in a link and a known bound on the length of time between the delivery of a request packet and the sending of a matching response packet. Then by controlling the interval between successive snapshot/reset phases it is easily possible to obtain a stabilizing snapshot protocol. The interval is chosen to be large enough such that all packets from the previous phase will have disappeared at the end of the interval [APV91a]. We call the general method timer flushing.

The main idea in timer flushing is to bound the lifetime of "old" state information in the network. This is done by using node clocks that run at approximately the same rate and by enforcing a maximum packet lifetime over every link. State information that is not periodically refreshed is "timed out" by the nodes. Timer flushing has been used in the OSI Network Layer Protocol [Per83] and the IEEE 802.1 Spanning Tree protocol [Per85]. Spinelli [Spi88b] uses timer flushing to build (practical) stabilizing Data Link and virtual circuit protocols.

In most real networks, each node sends "keep-alive" packets periodically on every link in order to detect failures of adjacent links. If no keep-alive packet arrives before a local timer expires, the link is assumed to have failed. Thus, it is common practice to assume time bounds for the delivery and processing of packets. Note also that the snapshot and reset packets used for local checking can be "piggy-backed" on these keep-alive packets without any appreciable loss in efficiency.

13 Summary

Self-stabilization was introduced by Dijkstra in a seminal paper [Dij74]. Dijkstra's shared memory model is enchanting in its simplicity and makes an ideal vehicle to describe non-trivial examples of self-stabilization. We showed how to simulate Dijkstra's model as one big IOA with a few restrictions. We also used Dijkstra's first two examples to introduce two powerful methods for self-stabilization: counter flushing and local checking and correction.

Next, we moved to a more realistic message passing model. We introduced a reasonable model for bounded storage links. Using this model, we formally defined the notions of local checkability and correctability and described the local correction theorem. This theorem shows that any locally correctable protocol can be stabilized in time proportional to the height of a partial order that underlies the definition of local correctability. Our proof of this theorem was constructive: we showed how to do augment the original protocol with snapshot and reset actions. We made the local snapshot and reset protocols stabilizing using counter flushing. While counter flushing relies on bounding the number of packets that can be stored on a link, we observed that a practical implementation would be based on bounding the time that a message could stay on a link – we called this technique timer flushing.

Together counter flushing, timer flushing, and local checking (with and without local correction) can be used to explain a number of self-stabilizing protocols and can be used to design new protocols. While these techniques cannot be used to solve every problem, there are a large number of useful protocols that can be efficiently stabilized using these notions. In [Var92a] we provide an example of a network reset protocol that – somewhat surprisingly - is locally correctable using the trivial partial order. In other places we have used local checking to provide solutions for mutual exclusion, the end-to-end problem, stabilizing synchronizers, and spanning tree protocols. More detailed references can be be found in [APV91b], [Var92b] and [AV91].

The messages used for local checking can be piggybacked on keepalive traffic in real networks without any appreciable loss of efficiency. As a side benefit, local checking also provides a useful debugging tool. Any violations of local checking can be logged for further examination. In a trial implementation of our reset procedure on the Autonet [MAM+90], local checking discovered bugs in the protocol code. In the same vein, local checking can provide a record of catastrophic, transient faults that are otherwise hard to detect.

References

[AB89] Yehuda Afek and Geoffrey Brown. Self-stabilization of the alternating bit protocol. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AGR92] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks. In *Proceedings of the 11th PODC*, Vancouver, British Columbia, August 1992.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, September 1990. Springer-Verlag (LNCS 486).

[APV91a] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Creating self-stabilizing protocols by using a reset protocol. Draft, 1991.

[APV91b] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[Awe90] Baruch Awerbuch. Shortest paths and loop-free routing in dynamic networks. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Philadelphia, PA*, September 1990.

[BSW69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.

[CRKG89a] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 224–236, 1989.

[CRKG89b] Chunhsiang Cheng, Ralph Riley, Srikanta P.R. Kumar, and Jose J. Garcia-Luna-Aceves. A loop-free extended Bellman-Ford routing protocol without bouncing effect. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 224–236. ACM SIGCOMM, ACM, September 1989.

[CSV89] Jeff Cooper, Robert Simcoe, and George Varghese. Stabilizing, hardware-based implementation of a flow control scheme for high speed links. Presented to ATM Forum in Aug 1993, January 1989.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[DIM91] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message driven protocols. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, Montreal, Canada, August 1991.

[Gar89] Jose J. Garcia-Luna-Aceves. A unified approach to loop-free routing using distance vectors or link states. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 212–223. ACM SIGCOMM, ACM, September 1989.

[KP90] Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[MAM+90] M.Schroeder, A.Birrell, M.Burrows, H.Murray, R.Needham, T.Rodeheffer, E.Sattenthwaite, and C.Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical Report 59, Digital Systems Research Center, April 1990.

[Per83] Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, December 1983.

[Per85] Radia Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proceedings of the the 9th Data Communication Symposium*, pages 44–53, September 1985.

[RF89] Balasubramanian Rajagopalan and Michael Faiman. A new responsive distributed shortest path routing algorihtm. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 237–246. ACM SIGCOMM, ACM, September 1989.

[Ros81] E. C. Rosen. Vulnerabilities of network control protocols: An example. *Computer Communications Review*, July 1981.

[Spi88a] John M. Spinelli. Reliable communication. Ph.d. thesis, MIT, Lab. for Information and Decision Systems, December 1988.

[Spi88b] John M. Spinelli. Reliable communication data links. Technical Report CICS-P-102, MIT, Lab. for Information and Decision Systems, December 1988.

[Var92a] George Varghese. Local checking and correction in a shared memory model. In preparation, 1992.

[Var92b] George Varghese. Self-stabilization by local checking and correction. Ph.D. thesis, to appear as a LCS Technical Report, 1992.