Self-Stabilization by Local Checking and Correction*

Baruch Awerbuch[†] Boaz Patt-Shamir[‡] George Varghese[§]

July 16, 1997

1 Introduction

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. Informally, a protocol is *self-stabilizing* if when started from an arbitrary global state it exhibits "correct" behavior after finite time.

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.

A self-stabilizing protocol copes with a set of failures that subsumes most previous categories, (such as link and node failures) and is robust against transient errors. Transient errors include memory corruption, as well as malfunctioning devices that send out incorrect packets. Transient errors do occur in real networks and cause systems to fail unpredictably [Ros81, Per83]. Thus stabilizing protocols are attractive because they offer *increased robustness*. In real networks, manual intervention has a high cost; a reasonable rule of thumb 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 self-stabilization offers a simple mechanism to handle a slew of faults. First, 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. Second, self-stabilization can subsume such common fault models as link and node failures. Most protocols have special mechanisms to deal with each anticipated fault, which complicates the protocol.

*A preliminary version of this paper appeared in the 32nd Proceedings of the IEEE Foundations of Computer Science (FOCS) Conference, 1991.

†Johns Hopkins University

‡Northeastern University

§Washington University, St. Louis, work done while at Lab. for Computer Science, MIT. Partially supported by an ONR Young Investigator Award.

A common criticism of the self-stabilization model is that 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; it is even harder to prevent a malfunctioning node from sending out incorrect messages.

A second criticism is that a self-stabilizing program 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.

Finally, self-stabilization is often contrasted with Byzantine fault models. Self-stabilizing protocols deal 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. In practice, self-stabilizing solutions are significantly cheaper than their Byzantine counterparts.

Previous Work: Self-stabilizing protocols were introduced by Dijkstra [Dij74]. They have been studied by various researchers (e.g., [BP89, GM90, KP90, DIM90, IJ90a, IJ90b, AG90, AKY90]). These papers describe self-stabilizing protocols for particular tasks such as token passing, spanning tree construction, and network reset.

In this paper, by contrast, we address the problem of constructing an efficient and general methodology for designing self-stabilizing protocols. We call our method *local checking and correction*. We state a general transformation technique that can take as input a certain type of protocol (that is locally checkable and correctable) and produce an equivalent stabilizing protocol. We show the power of our method by showing its application to several examples.

In the self-stabilization literature, there have been few general techniques for self-stabilization. In a seminal paper, Katz and Perry [KP90] showed how to compile an arbitrary asynchronous protocol into a stabilizing equivalent by *centralized checking and correction at a leader node*. The main technical difficulty in this approach is finding a self-stabilizing method to do checking and correction. For this purpose, [KP90] invented a self-stabilizing snapshot protocol. However, the need for centralized checking makes the performance of this approach rather poor.

By contrast, our method of local checking and correction divides the network into a number of overlapping *link subsystems* as shown in Figure 3. A link subsystem consists of a pair of neighboring nodes and the channels between them. In essence, we 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 and better performance.

2 Local Checking and Correction: A Preview

To make the notion of local checking and correction more concrete we quickly describe an example of a protocol that works between two nodes, a sender node and a receiver node (Figure 1). 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 1) 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

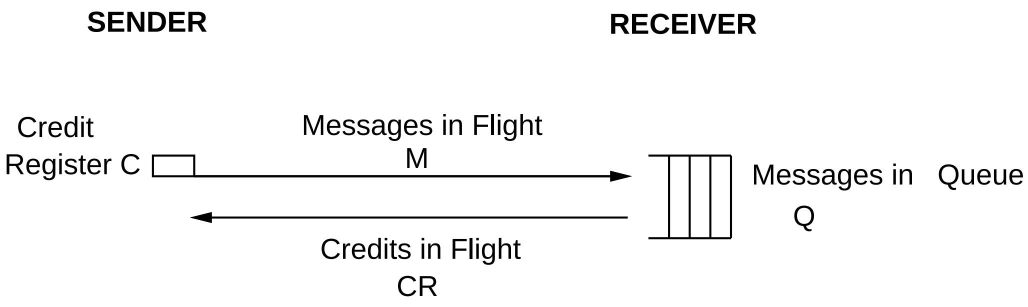

Figure 1: Credit Based Flow Control between a Sender and a Receiver

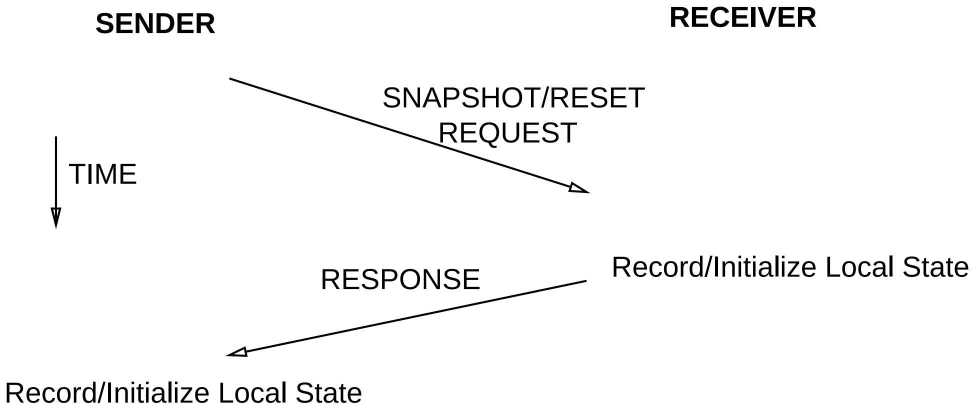

Figure 2: A Single Phase of Checking/Correction using Snapshots/Resets

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 1) 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: $C + M + Q + CR = Max$, a simple conservation invariant. 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 2) 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 2) 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 2) 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. Of course, the snapshot and reset protocols themselves have to be made self-stabilizing; we describe a simple technique for this purpose later in the paper.

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.[1]

2.1 Extending the Idea to a General Network

Briefly, the rest of this paper can be described as an extension (of the simple link checking and correction scheme described in the last subsection) to general network protocols. So consider a network as shown in Figure 3. 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 3. 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.

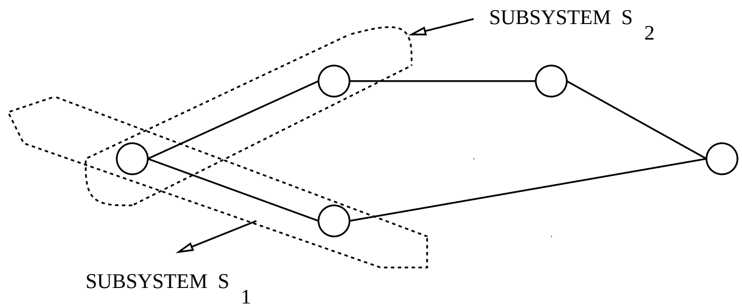

Figure 3: An Example of Two Overlapping Link Subsystems in a Network

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

[1] This fault-tolerant credit based scheme was proposed in [CSV89]. A variant of this scheme was proposed for hop-by-hop flow control on ATM links.

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 3) 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.

3 Paper Organization

We describe our model and stabilization definitions in Section 4. We then describe some key definitions of local checkability and correctability in Section 5. We use these definitions to state the local correction theorem in Section 6. The proof of this theorem is constructive and is spread over three sections. First, in Section 7 we describe the actual construction that underlies the proof of the theorem; in Section 8, we provide an intuitive justification of why this construction has the desired properties; and in Section 9 we outline a formal proof of the theorem based on the construction.

We then move to a discussion of the theorem's usefulness. Section 10 describes a complete example of an actual protocol that is made stabilizing using our theorem. Section 11 describes other theorems based on local checking that are also useful. Section 12 describes some how local checking can be implemented in real networks, and compares it to the notion of periodic state sending or soft-state that is being advocated for Internet protocol design[Cla91]. Finally, we state our conclusions in Section 13.

4 Model and Definitions

We first describe our modeling tool: timed I/O automata; next, we describe our definitions of stabilization; finally, we describe how we model network protocols using our tools.

4.1 Modeling Tool

We use the timed IOA model [MMT91] in which each action is associated with a fairness class c. To model asynchronous protocols, we assume that any continuously enabled action in class c will occur in time t_c protocols. We do not use lower bounds and we assume that the protocols must work regardless of the value of t_c used. The use of such parameterized upper bounds allows us to avoid the double effort of first proving correctness and then proving time bounds. Liveness arguments are replaced by showing time bounds on event occurrences. Our model is a special case of the timed I/O automaton model in [MMT91]. However, our terminology is slightly different from that of [MMT91].

An automaton is a state machine with transition labels called actions. When the automaton "runs" it produces an execution which is alternating series of states and actions starting with a start state. To model time, we associate upper bounds with the performance of continuously enabled state transitions of the automaton. We also associate time values with each action and state in an execution such that the these values are consistent with the specified upper bounds.

Formally, an *automaton* A consists of five components:

- a finite set of actions *actions*(A) that is partitioned into three sets called the set of *input, output, and internal* actions. The union of the set of input actions and the set of output actions is called the set of *external* actions. The union of the set of output and internal actions is called the set of *locally controlled* actions.

- A finite set of states called $states(A)$.
- A nonempty set $start(A) \subseteq states(A)$ of start states.
- A transition relation $R(A) \subseteq states(A) \times actions(A) \times states(A)$ with the property that for every state s and input action a there is a transition $(s, a, \tilde{s}) \in R(A)$.
- An equivalence relation $part(A)$ partitioning the set of locally controlled actions into equivalence classes, such that for each class c in $part(A)$ we have a positive real upper bound t_c. (Intuitively, t_c denotes an upper bound on the time to perform some action in class c.)

An action a is said to be *enabled* in state s of automaton A if there exist some $\tilde{s} \in states(A)$ such that $(s, a, \tilde{s}) \in R(A)$. An action a is *disabled* in state s of automaton A if it is not enabled in that state. Since one action may occur multiple times in a sequence, we often use the word *event* to denote a particular occurrence of an action in a sequence.

To model the passage of time we use a time sequence. A *time sequence* $t_0, t_1, t_2, \ldots$ is a non-decreasing sequence of non-negative real numbers; also the numbers grow without bound if the sequence is infinite. A *timed element* is a tuple (x, t) where t is a non-negative real and x is an element drawn from an arbitrary domain. A *timed state* for automaton A is a timed element (s, t) where s is a state of A. A *timed action* for automaton A is a timed element (a, t) where a is an action of A.

Let $X = (x_0, t_0), (x_1, t_1), \ldots$ be a sequence of timed elements. We will also use $x_j.time$ (which is read as the time associated with element x_j) to denote t_j. We say that element x_j occurs within time t of element x_i if $j > i$ and $x_j.time \leq x_i.time + t$. We will use $X.start$ (which is read as the start time of X) to denote t_0.

An *execution* α of automaton A is an alternating sequence of timed states and actions of A of the form $(s_0, t_0), (a_1, t_1), (s_1, t_1), (a_2, t_2), (s_2, t_2), \ldots$ such that the following conditions hold:

1. $s_0 \in start$ and $(s_i, a_{i+1}, s_{i+1}) \in R$ for all $i \geq 0$.
2. The sequence can either be finite or infinite, but if finite it must end with a timed state.
3. The sequence $t_0, t_1, t_2, \ldots$ is a time sequence.
4. If any action in any class c is enabled in any state s_i of α then within time $s_i.time + t_c$ either some action in c occurs or some state s_j occurs in which every action in c is disabled.

Notice that the time assigned to any event a_i in α (i.e., $a_i.time$) is equal to the time assigned to the next state (i.e., $s_i.time$). Notice also that we have ruled out the possibility of so-called "Zeno executions" in which the execution is infinite but time stays within some bound.

Let α be an execution. Let γ be the subsequence of α consisting of timed external actions, and let t_0 be the start time of α. The *behavior* β corresponding to α is the sequence $\beta = t_0, \gamma$. The behaviors of automaton A are the behaviors corresponding to the executions of A. We will use $X.start$ to denote the starting time of either a behavior or or an execution X. Notice that start times are allowed to be *arbitrary non-negative* real numbers.

There is a notion of *composition* [MMT91] that produces a composite automaton $A = \Pi A_i$ out of a set of compatible constituent automata A_i. In a composition, input and output actions of the same name are performed simultaneously; the state of the composite automata is the composition of the constituent automata states. We use $\beta | A_i$ to represent the projection of a behavior β of A on to some constituent automaton A_i. We assume that $\beta | A_i$ inherits the times of β in the natural way. Our definitions of composition are identical to that in [MMT91].

4.2 Stabilization: Definitions and Properties

We begin with a standard state-based definition of stabilization and then describe a new definition of stabilization in terms of external behaviors.

Definitions of Stabilization based on Executions: Intuitively, if automaton A stabilizes to the executions in set C in time t, then within time t, all executions of A begin to "look like" an execution in set C. We formalize this by defining a t-suffix of an execution α. Intuitively, this is a suffix of α whose first element occurs no more than time t after the start of α. We say that α' is a *t-suffix of execution α* if: $\alpha'.start - \alpha.start \leq t$ and α' is a suffix of α. We can now define execution stabilization.

Let C be a set of sequences of timed elements. We say that automaton A *stabilizes to the executions* in C in time t if for every execution α of A there is some t-suffix of execution α that is in C. We also say that automaton A *stabilizes to the executions* of automaton B in time t if for every execution α of A there is some t-suffix of execution α that is an execution of B. Execution stabilization is transitive; this allows us to prove execution stabilization in several stages.

Lemma 4.1 *If automaton A stabilizes to the executions of automaton B in time t_1 and B stabilizes to the executions of automaton C in time t_2, then A stabilizes to the executions of C in time $t_1 + t_2$.*

Definitions of Stabilization based on External Behavior: A major theme of the I/O Automaton model [LT89] is the focus on external behaviors of an automaton for specifying correctness. Thus it natural to look for a definition of stabilization in terms of external behaviors. Typically, the correctness of an IOA is specified by a *problem* or set of legal behaviors P. An IOA A is said to *solve* problem P if the behaviors of A are contained in P. For stabilization, however, we weaken this definition and ask only that an automaton exhibit correct behavior after some finite time.

As in the case of execution stabilization, we begin with the definition of a t-suffix of a behavior β. Intuitively, this is a portion of β that starts at time no more than t after the start of β. However, this is not as easy as defining a t-suffix of an execution because a behavior $\beta = (t_0, \gamma)$ consists of two components: a start time t_0 and a sequence of timed actions γ. We cannot simply define a t-suffix of β to be a suffix of β; the t-suffix must have a start time as well as a sequence of timed actions.

Formally, consider any two behavior sequences $\beta = t_0, \gamma$ and $\beta' = t_0', \gamma'$. We say that β' is a *t-suffix of behavior β* if:

- $\beta'.start - \beta.start \leq t$.
- γ' is a suffix of γ containing all actions in β that occur at times strictly greater than $\beta'.start$.

Using this, we can define behavior stabilization analogous to execution stabilization (our definition is adapted from a definition suggested by Nancy Lynch.) Let P be a set of behavior sequences. An IOA A *stabilizes to the behaviors* in P in time t if for every behavior β of A there is a t-suffix of behavior β that is in P. An automaton A is said to *stabilize to the behaviors* of another automaton B in time t if for every behavior β of A there is a t-suffix of behavior β that is a behavior of B.

The following lemmas follow from the definitions. The transitivity lemma allow us to prove behavior stabilization results in stages. All lemmas described in this subsection are intuitive and so we omit proofs (which can be found in [Var93]).

Lemma 4.2 *If automaton A stabilizes to the behaviors of automaton B in time t_1 and B stabilizes to the behaviors of automaton C in time t_2, then A stabilizes to the behaviors of C in time $t_1 + t_2$.*

Lemma 4.3 *If automaton A stabilizes to the behaviors of an automaton B in time t and $\tilde{t} \geq t$ then A stabilizes to the behaviors of B in time $\tilde{t}$.*

The previous lemma motivates a natural complexity measure called the *stabilization time* from A to B. Intuitively, this is the smallest time after which we are guaranteed that A will behave like B. Formally, the *stabilization time* from A to B is the infimum of all t such that A stabilizes to the behaviors of B in time t.

The next lemma shows that execution stabilization implies behavior stabilization. Thus to *prove* a behavior stabilization result, we first prove a corresponding execution stabilization result. Behavior stabilization is typically used for *specification* and execution stabilization is used for *proofs*.

Lemma 4.4 *If automaton A stabilizes to the executions of automaton B in time t then automaton A stabilizes to the behaviors B in time t.*

The following useful notation allows us to specify an IOA that is identical to another IOA except for start states. For any automaton A and any subset L of $states(A)$, we denote by $A|L$ the automaton that is identical to A except that $start(A|L) = L$.

We will mostly deal with stabilization properties of a special class of automata called *unrestricted automata (UIOA)*. Intuitively, a UIOA models a system that can start in an arbitrary state. Formally, a UIOA A is an automaton such that $start(A) = states(A)$.

4.3 Network Model

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 also 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. In practice, these are not restrictive assumptions because real links can only store a bounded number of packets. Also real network nodes typically have a unique node identifier; the lower ID node on every link can then become the "leader".

We now describe how we model network topology, network links, and network nodes.

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

Modeling 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. Fortunately, this is perfectly consonant with real networks whose links indeed have bounded storage equal to the product of the propagation delay and transmission speed.

Each p belongs to the packet alphabet P defined above.
The state of the automaton consists of a single variable $Q_{u,v} \in P \cup nil$.

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

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

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 4: Unit Storage Data Link automaton

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. We will argue that a UDL can be implemented over real physical channels.

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.

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 4.

$C_{u,v}$ 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 (SEND$_{u,v}(p)$), an action to receive a packet at node v (RECEIVE$_{u,v}(p)$), and an action 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.

Modeling 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 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.

We wish the interface to a UDL to stabilize to "good behavior" even when the sender and link begin in arbitrary states. Arbitrary start states can cause two problems. First, if *free* = *true* but the UDL contains a packet, then the first packet sent by the sender can be dropped. However, 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 continuously enables the Free action whenever the link does not contain a packet.

Thus we will require that each sending node u keep a corresponding $free_u[v]$ variable for each neighbor v. By our convention, u can only send packets to v when $free_u[v] = true$. Thus we will also require that u enqueue packets that it wants to send to v in an outbound queue for the link called $queue_u[v]$. When $free_u[v]$ becomes *true*, the packet at the head of the outbound queue is sent on the link. The use of the queuing and free disciplines is quite natural for a UDL; more importantly, these conventions make it easy to transform a node automaton to do local checking. This will become clearer in a few sections.

The preceding paragraphs motivate the following formal definition.

We say that an automaton N is a node automaton for node u in graph $G = (V, E, l)$ and with node delay t if:

- N has an output action $\textsc{Send}_{u,v}(p)$ and input actions $\textsc{Receive}_{v,u}(p)$ and $\textsc{Free}_{u,v}$ for each $p \in P_{data}$ and for each neighbor v of u.
- N has a boolean variable $free_u[v]$ and a queue variable $queue_u[v]$ for every v such that $(u, v) \in E$. The queue variable $queue_u[v]$ is a queue of bounded size consisting of packets drawn from the data packet alphabet P_{data}.[2]
- Actions of N other than $\textsc{Send}_{u,v}(p)$ can only change $queue_u[v]$ by adding packets (drawn from alphabet P_{data}) to the tail of $queue_u[v]$. Of course, such actions can leave the queue unchanged.
- The code for the output action $\textsc{Send}_{u,v}(p)$ and the input action $\textsc{Free}_{u,v}$ at node N is as shown in Figure 5.
- Each $\textsc{Send}_{u,v}(p)$ action in N is in its own class with upper bound called the *node delay* that is equal to t for all classes.

In particular, note that every $\textsc{Send}_{u,v}(p)$ action at a node is a locally controlled action. Also, in all the network automata described in this paper, the transitions in automaton N_u will only depend on the

[2] The reader may be puzzled by the fact that the links accept both data and control packets but the node automata only send data packets. In a few sections, we will create *augmented node automata* by adding actions to ordinary node automata. These extra actions are used to send and receive control packets for the purposes of local checking/correction.

$\textsc{Send}_{u,v}(p)$ (*output action to send packet at head of outbound queue*)
  Precondition: $free_u[v] = true$ and p is head of $queue_u[v]$
  Effect: $free_u[v] := false$; Remove p from head of $queue_u[v]$

$\textsc{Free}_{u,v}$ (*input action*)
  Effect: $free_u[v] := true$; (*record that link is free*)

Figure 5: Code at a node automaton to send a data packet and to respond to a free signal from a link

state of u, the leader function l, and the identities of the neighbors v of u. In other words, N_u will use only local information available to u about the graph G. We prefer not to formalize this requirement as part of the definition of a network automaton. Instead, we will use this as an informal criterion to rule out trivial solutions (to some problems) in which N_u encodes the entire graph.

Network Automata: Now we are ready to define a network automaton. Naturally, it is the composition of a set of node and channel automata.

Let $G = (V, E, l)$ be a topology graph. Let N be a set containing exactly one node automaton N_u, for every $u \in V$ and such that each N_u has node delay t. Let $C_{u,v}$ be a UDL for each $(u, v) \in E$ such that every UDL has link delay $\tilde{t}$. Then $Net(G, N, t, \tilde{t})$, the network automaton with node delay t and link delay $\tilde{t}$, is the composition of the automata N_u for all $u \in V$ with the automata $C_{u,v}$ for all $(u, v) \in E$.

For the most part we will deal with network automata in which the node and link delays are fixed. Thus we let t_n denote the *default node delay* and t_l denote the *default link delay*. If we do not explicitly mention the node and link delays, then the node and link delays are assumed to be t_n and t_l respectively. Thus we will use $Net(G, N)$ to denote $Net(G, N, t_n, t_l)$. We will sometimes say that a time t is a *constant* if $t = c_1 t_n + c_2 t_l$, where c_1 and c_2 are some real scalar constants. We use "constant time" to emphasize that the time does not depend on the size of the network but only on the node and link delays.

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]).

The sender end of the Data Link protocol has a queue that can contain a single packet. When the queue is empty, the $\textsc{Free}$ signal is enabled. When a $\textsc{Send}(p)$ 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. We can now use any stabilizing Data Link protocol (e.g., [AB89]).

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.

5 Local Checking and Correction

In this section, we introduce formal definitions of local checkability and local correctability.

5.1 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 $\mathcal{N} = Net(G, N)$. 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 describes what it means for a local property to hold in a state s of the entire automaton. 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: 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.

$\mathcal{L}$ 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 $L_{u,v} \in \mathcal{L}$.

The *conjunction* of a link predicate set $\mathcal{L}$ is the predicate $\{s : s \text{ satisfies } L_{u,v} \text{ for all } L_{u,v} \in \mathcal{L}\}$. 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.

5.2 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 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.

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 $L \supseteq \mathit{Conj}(\mathcal{L})$.
- 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.

5.3 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. 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:

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.

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

6 Local Correction Theorem

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. We show how to take a network automaton $\mathcal{N}$ that can be locally corrected to L, and *transform* it into a new automaton $\mathcal{N}^+$. The new automaton $\mathcal{N}^+$ has the property that in all its executions, property L holds after a bounded amount of time. More precisely, $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}|L$ in a bounded amount of time. The next subsection contains a formal statement of this result.

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.[3]

[3]Without this assumption, we have to complicate the code and proof to deal with simultaneous checking and correction actions by both ends of a link. This can actually be done, thereby getting rid of the requirement for a leader on links. But it isn't worth the increased complexity.

To state the result formally, we need the following definitions. First, when we augment a network automaton the resulting automaton should have the same topology and also be an unrestricted automaton (UIOA) that can start in any state. The topology restriction rules out trivial "centralized" solutions. We also require that the links remain UDLs. To formalize these requirements, we define a new type of automaton.

Let $G = (V, E, l)$ be a topology graph. An *automaton for graph* G is the composition of an automaton for each $u \in V$, together with $C_{u,v}$ for each $(u, v) \in E$. We assume that all automata being composed are compatible.

Notice that any network automaton for graph G is also an automaton for graph G. However, an automaton for graph G need not be a network automaton because a network automaton has additional constraints (such as having outbound queues and free variables for each link) on the node automata.

A UIOA for graph G is an automaton for graph G that is also a UIOA. Recall that we used $A|L$ to denote the automaton identical to A except that its start states belong to set L. The following piece of shorthand is useful for a concise statement of the theorem.

Let $\mathcal{N}$ denote a network automaton. We will use $\mathcal{N}(t)$ to denote the automaton that is identical to $\mathcal{N}$ except that the link and node delays in $\mathcal{N}(t)$ are equal to t.

Now (finally!) we can state our theorem. Intuitively, it 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 $\mathcal{N}^+$ such that $\mathcal{N}^+$ satisfies the following property: in time proportional to *height*$(<)$, every behavior of $\mathcal{N}^+$ will "look like" a behavior of $\mathcal{N}$ in which L holds and in which the node and link delays are increased by some constant factor.

Theorem 6.1 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 for graph* G *and constants* c *and* $\tilde{c}$ *such that* $\mathcal{N}^+$ *stabilizes to the behaviors of* $\mathcal{N}(c)|L$ *in time* $\tilde{c} \cdot height(<)$.

7 Transformation Code used for Proof of Local Correction Theorem

The proof of the Local Correction theorem is constructive. We will actually describe how to transform $\mathcal{N}$ into a $\mathcal{N}^+$ that satisfies the theorem statement. Rather than present the formal transformation directly, we first describe an overview of the transformation code. We then describe the formal transformation.

7.1 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.

For those familiar with snapshot protocols, the structure of our local snapshot protocol is slightly different from the well-known Chandy-Lamport snapshot protocol [CL85]. It is easy to show that the Chandy-Lamport scheme cannot be used without modifications over unit storage links. Briefly, the reason is as follows. The correctness proof of the algorithm in [CL85] is based on reordering executions while preserving causality constraints. The only causality constraint for a link in [CL85] is that any action that sends a packet p on link L cannot be reordered to come after an action that receives p on link L. However,

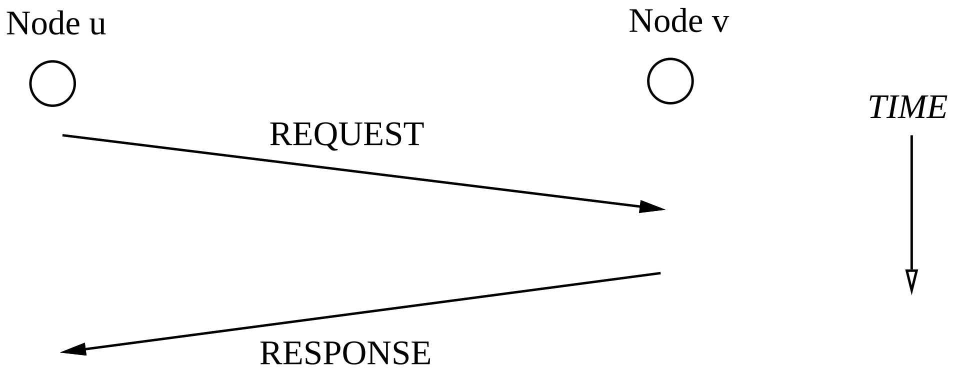

Figure 6: The structure of a single phase of the local snapshot/reset protocol

a UDL has an additional causality constraint. A free signal delivered by link L after delivering packet p cannot be reordered to come before the action that delivers packet p. The Chandy-Lamport scheme was not designed to incorporate this extra causality constraint. As a result, if the Chandy-Lamport scheme is used unmodified over a network with UDLs, the snapshot may (incorrectly) return a state in which there is more than one packet on a UDL!

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 6.

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 ($phase_u[v]$) 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 *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 7 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 7.

To make the snapshot and reset protocols stabilizing, we use what we call counter flushing.[Var94].

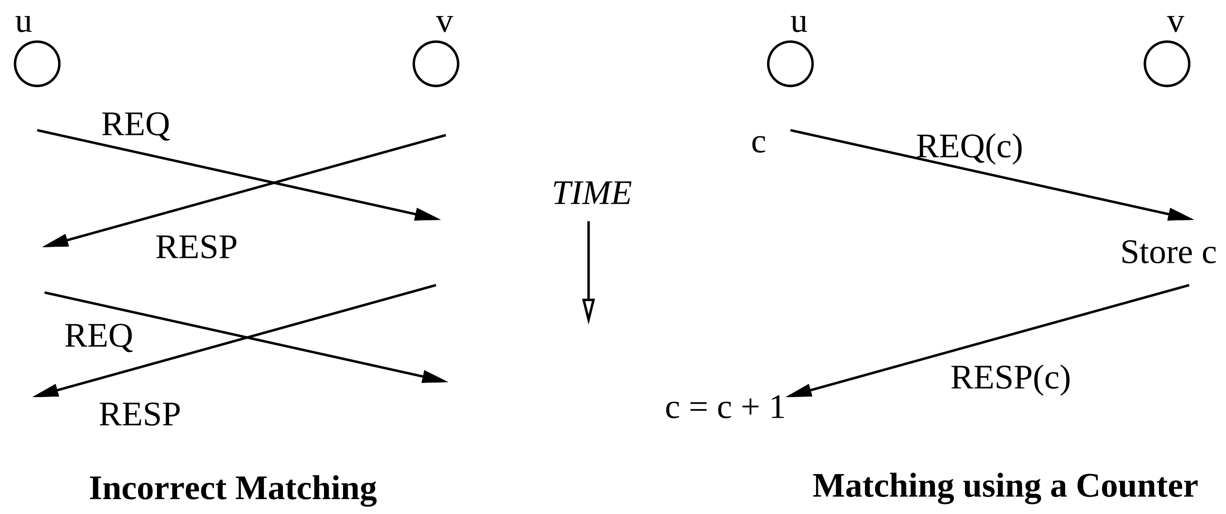

Figure 7: Using counter flushing to ensure that request-response matching will work correctly within a small number of phases.

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 $phase_u[v]$ 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.

7.2 Constructing Augmented Automata: Formal Description

To transform $\mathcal{N}$ into $\mathcal{N}^+$ we will show how to transform each node automaton N_u in $\mathcal{N}$ into a new, augmented node automaton N_u^+. Finally, $\mathcal{N}^+$ is the composition of the new node automata and the (unchanged) channel automata.

Assume that network automaton $\mathcal{N} = Net(G, N)$ can be *locally corrected* to L using link predicate set $\mathcal{L} = \{L_{u,v}\}$ and local reset function f. We create $\mathcal{N}^+ = Augment(\mathcal{N}, \mathcal{L}, f)$ by adding states and actions to each node automaton N_u as follows:

- We add the following new variables and their domain specifications, to the state of N_u, one for each neighbor v of u as shown below:.
  - $count_u[v] \in \{0 \dots 3\}$ (used to number request and response packets to ensure proper matching.

The magic number 3 arises because a link subsystem can store at most three distinct counter values on the sending link, receiving link, and at the receiver node.)

  - $mode_u[v] \in \{reset, snapshot\}$ (used to keep track of whether the current phase is a snapshot or reset phase.)
  - $phase_u[v]$ is a Boolean (set to true during a reset or snapshot phase. It is used by the leader node to inhibit data packets from being sent during a phase.)
  - $freeq_u[v]$ is a Boolean (set to true after *any* packet is sent and set to false after any packet is removed from the link. The original variable $free_u[v]$ will be set to true after a data packet is sent and set to false after a data packet is removed from the link. We could have optimized by using just one free variable but keeping two variables makes the projection and the proof easier.)
  - $turn_u[v] \in \{response, data\}$ (used to keep track of whether a data packet or a response packet has the next turn to be sent.)

- We add two new packet types to $\mathcal{N}^+$. Recall that there are two basic types of packets, data packets and control packets. We will encode request and response packets as control packets. We will use the symbols p_{data}, p_{req}, and p_{resp} to denote data, request and response packets respectively. The format of a data packet is defined by automaton $\mathcal{N}$. The encoding of the other two packets is:

  Request : (*Control*, *Request*, *count*, *mode*), where *count* is an integer from $0 \ldots 3$, and *mode* is either *reset* or *snapshot*.

  Response : (*Control*, *Response*, *count*, *mode*, *node_state*), where *count* is an integer from $0 \ldots 3$, *mode* is either *reset* or *snapshot*, and *node_state* is a state of a node automaton N_u in $\mathcal{N}$.

  As usual, we use record notation to extract fields of a packet. Thus $p_{req}.count$ is the *count* field in a request packet.

- We modify the $\textsc{Send}_{u,v}(p)$ (for data packets p) and $\textsc{Free}_{u,v}$ actions of the original automaton N_u as shown in Figure 8. This code contains modifications for both the leader and responder ends in one piece of code; however, some parts of the code applies only to leaders and some parts only apply to responders.

  If node u is the leader on link (u, v), then we enable sending data packets only when $phase_u[v] = false$ indicates a phase is not in progress. After the data packet is sent, we set $phase_u[v] = true$. If node u is not the leader on link (u, v), then we enable sending data packets only when $turn_u[v] = data$. Also immediately after sending a data packet, we set $turn_u[v] = response$, which allows the response packets to get a fair turn. We use the $freeq_u[v]$ variable to keep track of whether there is some packet (either data or control) on $C_{u,v}$ while $free_u[v]$ keeps track of whether there is a data packet on $C_{u,v}$. The code appears to have some redundant checks (for example, the code checks both free variables before sending a data packet) but these extra checks do make the proof easier.

- We add the actions shown in Figures 9 and 10 to N_u for each neighbor v of u. These actions only apply to control packets. We use the following notation. Let s denote the current state of $\mathcal{N}^+$ when an action is performed. Let $s|u$ denote the current state of $\mathcal{N}_u^+$ *projected onto the original automaton* N_u. In order to project s to $s|u$ we do the following. All variables of N_u take the same values as the corresponding variables in N_u^+.

  When we apply f to the projected state by setting $s|u$ to $f(s|u, v)$ we affect only the projected variables. Thus, for instance, the value of $freeq_u[v]$ remains unchanged.

$\text{SEND}_{u,v}(p)$ (*output action for $p \in P_{data}$ only*)
Preconditions:
$freeq_u[v] := true$ and $free_u[v] := true$
p is head of $queue_u[v]$
$((l(u,v) = u)$ and $(phase_u[v] = false))$ OR $((l(u,v) = v)$ and $turn_u[v] = data))$
Effect:
$freeq_u[v] := false$ and $free_u[v] := false$;
Remove p from head of $queue_u[v]$
$turn_u[v] = response$ (* give response packets a turn; only affects responders*)
$phase_u[v] = true$ (* start a new checking/correction phase; only affects leaders*)

$\text{FREE}_{u,v}$ (*input action*)
Effect: $freeq_u[v] := true$ and $free_u[v] := true$;

Figure 8: Code for the modified $\text{SEND}_{u,v}(p)$ actions at a modified node N_u^+.

- We add two extra classes for every neighbor v of u to N_u. Each output action of the form $\text{SEND}_{u,v}(Control, Request, *)$ is added to a *new* partition class. Similarly, each output action of the form $\text{SEND}_{u,v}(Control, Response, *)$ is added to a *new* partition class. The time associated with all new classes is still the node delay t_n.
- We hide all actions of $\mathcal{N}^+$ that are not actions of $\mathcal{N}$.

8 Intuitive Proof of Local Correction Theorem

An outline of a formal proof of the Local Correction theorem is described in the next section. A few missing details can be found in [Var93]. 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 11 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 11.

The snapshot 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 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

```
SEND_{u,v}(p_req)            (*output action: u repeatedly sends a request till it gets a response*)
   Precondition:
      l(u,v) = u                                          (*u is the leader of link subsystem*)
      (phase_u[v] = true) or (queue_u[v] is empty)    (*phase in progress or no data packets waiting*)
      freeq_u[v] = true                                   (* no packet in transit on link to v *)
      p_req.count = count_u[v];                           (*count in packet is count of phase*)
      p_req.mode = mode_u[v];                             (*mode in packet is mode of phase*)
   Effect:
      freeq_u[v] = false                                  (* set to false until link says it is free*)
      phase_u[v] := true;                                 (*remains true until matching response returns*)


RECEIVE_{v,u}(p_req)                                      (*input action, receive request at u from v*)
   Effect:
      If p_req.count ≠ count_u[v] and l(u,v) = v then     (* not a duplicate or invalid packet*)
         count_u[v] := p_req.count;                       (*remember count*)
         mode_u[v] := p_req.mode;                         (*remember mode*)
```

Figure 9: Code to send and receive request packets at node u.

```
SEND_{u,v}(p_resp)            (*output action: u repeatedly sends a response to last request*)
   Precondition:
      l(u,v) = v                                          (*u is not the leader of link subsystem*)
      (turn = response) or (queue_u[v] is empty)      (*response's turn or no data packets waiting*)
      freeq_u[v] = true                                   (* no packet in transit on link to v *)
      p_resp.count = count_u[v];
      If mode_u[v] = snapshot then p_resp.node_state = s|u else p_resp.node_state := f(s|u,v)
   Effect:
      If mode_u[v] = reset then s|u := f(s|u,v)           (*reset node u's state locally*)
      mode_u[v] := snapshot                               (* return to default mode of snapshot*)
      turn_u[v] := data                                   (*give data packets a turn*)
      freeq_u[v] := false                                 (* set to false until link says its free*)


RECEIVE_{v,u}(p_resp)                                     (*input action to receive response at u from v*)
   Effect:
      If (count_u[v] = p_resp.count) and (phase_u[v] := true) and (l(u,v) = u) then
         If mode_u[v] = snapshot then
            If (s|u, nil, nil, p_resp.node_state) ∉ L_{u,v} then mode_u[v] := reset
         Else if mode_u[v] = reset then s|u = f(s|u,v)    (*reset node u's state locally*)
         phase_u[v] := false;                             (*end of phase*)
         count_u[v] := (count_u[v] + 1) mod 4;
```

Figure 10: Code to send and receive response packets at node u.

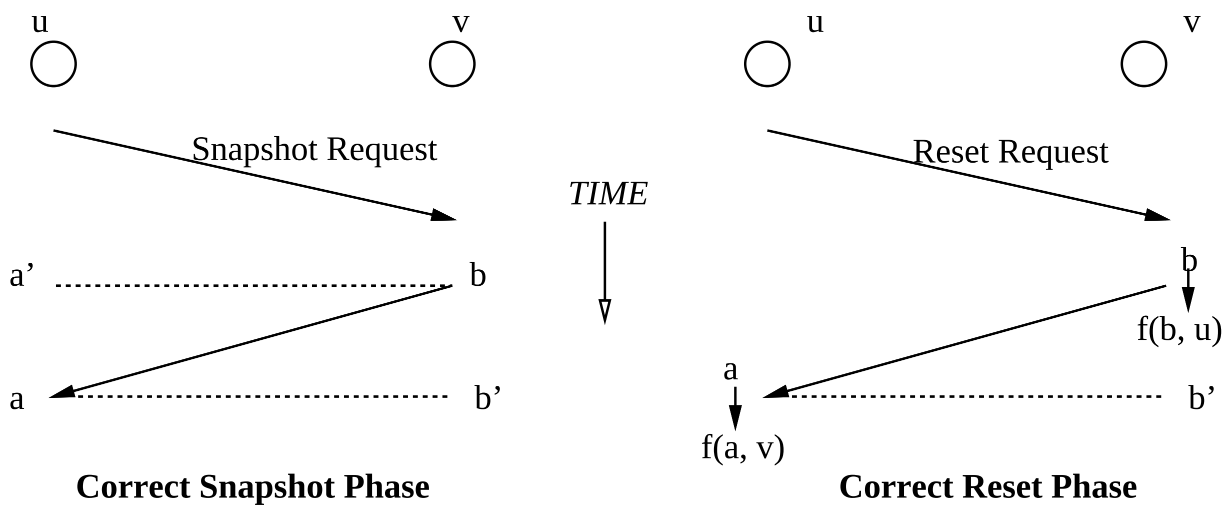

Figure 11: Local Snapshots and Resets work correctly if requests and responses are properly matched.

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 11 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 11.

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 $\{u,v\} \not< \{w,x\}$ 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 Formal Proof of Local Correction Theorem

There are a number of details glossed over in the intuitive proof that are spelled out in the formal proof. We only outline the major steps in the proof here; the complete details can be found in [Var93].

9.1 Proof Techniques

The following lemma is immediate from the definitions. (Recall that a problem P is a set of behaviors of an automaton.)

Lemma 9.1 *Consider an automaton A, $L \subseteq states(A)$, and a problem P. Suppose the following conditions are true:*

- *A stabilizes to the behaviors of $A|L$ in time t.*
- *Any behavior of $A|L$ is in P.*

Then A stabilizes to the behaviors in P in time t.

To prove that any behavior of some automaton A is a behavior contained in some problem P, it suffices to prove that every behavior of A is a behavior of some other automaton B, and that every behavior of B is in P. We now describe two techniques one can use for proving the two items on the list.

Proving that an Automaton Solves a Problem: To prove that any behavior of some automaton A is a behavior contained in some problem P, it suffices to prove that every behavior of A is a behavior of some other automaton B, and that every behavior of B is in P.

There are well-known techniques (e.g., [LT89]) to show any behavior of an automaton A is a behavior of another automaton B. A commonly used technique is a refinement mapping. The basic idea is to establish a suitable mapping f between a state of A and a state of B. Given an execution α of A, we use f to construct a mapped execution of B that has the same external behavior as α.

Theorem 9.2 Refinement Mapping: *Let A and B be automata with the same set of external actions. Let t_c be the upper bound on the time to perform actions in any class c of B. Let f be a mapping from the states of A to the states of B such that:*

1. *For any start state s of A, $f(s)$ is a start state of B.*
2. *For all transitions $(s, \pi, \tilde{s})$ of A, either*
   - *π is an action of B and $(f(s), \pi, f(\tilde{s}))$ is a transition of B OR*
   - *π is not an action of B and $f(s) = f(\tilde{s})$.*
3. *For any class c of B and any execution α of A, suppose some action in c is enabled in $f(s)$. Then within t_c time of s in α either:*
   - *Some action in class c occurs OR*
   - *Some state $\tilde{s}$ occurs such that no action in class c is enabled in $f(\tilde{s})$.*

Then every behavior of A is a behavior of B

Proof: See [Var93]. ∎

Proving that an Automaton Stabilizes to another Automaton: We give one such technique in the following definition and theorem. The technique is similar to techniques used for proving liveness properties (e.g., [OL82, MP91]) of concurrent programs.

Let L be a closed predicate of automaton A – once L is true in an any execution of A, L remains true for the rest of the execution. We would like to prove that in any execution of A, L becomes (and stays) true in bounded time. This implies that A stabilizes to the executions of $A|L$ in bounded time. We will describe a proof rule for this purpose. Intuitively, instead of proving directly that the goal L eventually holds we prove that a number of subgoals L_i (each of which is a predicate of A) become and stay true. The L_i are chosen so that L is true if all the individual L_i are true.

Each subgoal L_i can be intuitively thought of as "depending" on some other set of subgoals. This dependence relation is formalized by a partial order $<$. If $j < i$, then (intuitively) L_i "depends" on L_j. Suppose we could show that if all the subgoals that L_i depend on become and stay true, then L_i becomes and stays true. Then we can conclude that eventually all subgoals (and hence L) become and stay true. Intuitively, this follows because the dependency relation is *acyclic* since it is formalized using a partial order.

We can extend this idea to a timed setting to show that L will become true in time proportional to the maximum length of a "dependency chain". A dependency chain is formalized using the standard concept of a *chain* in a partially ordered set. Consider a set $\{L_i, i \in I\}$ where $(I, <)$ is a finite partially ordered set. A chain is simply a sequence $L_1 < L_2 < \ldots < L_i$.

The preceding discussion motivates the following definition and theorem.

We say that an automaton A is *stabilized* to predicate L using predicate set $\mathcal{L}$ and time constant t if:

1. $\mathcal{L} = \{L_i, i \in I\}$ of sets of states of A, where $(I, <)$ is a finite, partially ordered index set. We let *height*$(\mathcal{L})$ denote the maximum length chain in the partial order.

2. $\cap_{i \in I} L_i \subseteq L$.

3. For all $i \in I$ and for all steps $(s, \pi, \tilde{s})$ of A, if s belongs to $\cap_{j \leq i} L_j$, then $\tilde{s}$ belongs to L_i.

4. For every $i \in I$ and every execution α of A and every state s in α the following is true. Suppose that either $s \in \cap_{j<i} L_j$ or there is no $L_j < L_i$. Then there is some state $\tilde{s} \in L_i$ that occurs within time t of s in α.

The first condition says there is a partial order on the predicates in $\mathcal{L}$. The second says that L becomes "true", when all the predicates in $\mathcal{L}$ become true. The third is a stability condition. It says that any transition of A leaves a predicate L_i true, if all the predicates less than or equal to L_i are true in the previous state. Finally the last item is a liveness condition. It says that if all all the predicates *strictly* less than L_i are true in a state, then within time t after this state, L_i will become true.

We define *height*(L_i), the height of a predicate $L_i \in \mathcal{L}$, to be the maximum length of a chain that ends with L_i in the partial order. The value of *height*$(\mathcal{L})$ is, of course, the maximum height of any predicate $L_i \in \mathcal{L}$. By the liveness condition, within time t all predicates with height 1 become true; these predicates stay true for the rest of the execution because of the third stability condition. In general, we can prove by induction that within time $i \cdot t$ all predicates with height i become and stay true. This leads to a simple but useful theorem:

Theorem 9.3 Execution Convergence: *Suppose that automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t. Then, A stabilizes to the executions of $A|L$ in time height$(\mathcal{L}) \cdot t$.*

Proof: See [Var93]. ∎

9.2 Overview of Formal Proof

We wish to prove the local correction theorem, Theorem 6.1. Thus we have to prove that in time proportional to *height*$(<)$, every behavior of $\mathcal{N}^+$ will "look like" a behavior of $\mathcal{N}$ in which L holds and in which the node and link delays are increased by some constant factor.

The formal proof is based on the proof technique described in in Lemma 9.1. Thus the proof consists of two major parts:

- We first define a predicate Q (more details below) and show that any execution of $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$. We prove this using the Execution Convergence theorem, Theorem 9.3.

- We show that any behavior of $\mathcal{N}^+|Q$ is also a behavior of $\mathcal{N}(c)|L$, for some constant c. We prove this using the Refinement Mapping theorem, Theorem 9.2

We now present a more detailed roadmap of each part of the formal proof.

The first part of the proof is described in Sections 9.3 to 9.5. To show that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$ we use another two step process:

- In Section 9.4 we formally define the concept of a clean link alluded to earlier. Intuitively, a link is clean if the snapshot numbering scheme is working correctly, meaning that requests and responses will properly be matched. We use the predicate C to denote the fact that all links are clean. At the end of Section 9.4, in Theorem 9.9 we prove that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$ in constant time. The proof is based on the counter flushing intuition we have described earlier.

- In Section 9.5 we define another important concept called a quiet link. Recall that our intent is to make the local predicate $L_{u,v}$ hold for every link. Intuitively, a link (u, v) is quiet if two properties hold. First, $L_{u,v}$ holds. Second, if all links less than (u, v) in the partial order are also quiet, then there will be no more reset actions on that link. We use the predicate Q to denote the fact that all links are quiet. At the end of Section 9.5, in Theorem 9.14 we prove that $\mathcal{N}^+|C$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time proportional to the height of the partial order.

The second part of the proof (Section 9.6, (Lemma 9.15)) shows that any behavior of $\mathcal{N}^+|Q$ is also a behavior of $\mathcal{N}(c)|L$, for some constant c. This is done using the Refinement Mapping theorem, Theorem 9.2. In order to use this theorem we must derive a projected state of $\mathcal{N}$ from a state of $\mathcal{N}^+$. We have already seen how to derive a projected state $s|u$ of a node N_u from a state s of N_u^+. To complete our job, we need to state how to project the state of the channels in $\mathcal{N}^+$ to $\mathcal{N}$.

Consider the problem of projecting the channel state. Intuitively, in $\mathcal{N}^+$, the original automaton $\mathcal{N}$ is "sharing" each channel with request and response packets. Let's look at the behavior of $\mathcal{N}^+$. When a control packet is on the channel $C_{u,v}$, we can pretend that, as far $\mathcal{N}$ goes, the channel is really empty. There are two consequences of this. First, if the projected N_u has a data packet to send, it can take longer before the data packet is sent. Second, it can take longer before a free signal is delivered by the link. This can happen if there is a control packet on the link. We model this by saying that in the projected behavior *the node and link delays are increased by a constant factor.* This is the source of the increased link and node delays in Theorem 6.1.

For any state s of $\mathcal{N}^+$, we define $Proj(s)$, the state of $\mathcal{N}^+$ projected onto $\mathcal{N}$, as follows:

- For any two neighboring nodes u and v: if $s.Q_{u,v} \in P_{data}$ then $Proj(s).Q_{u,v} = s.Q_{u,v}$; else $Proj(s).Q_{u,v} = nil$. (i.e., if there is a data packet p in channel $C_{u,v}$ in the original state, then p is also present in the projected state; if not, $C_{u,v}$ is considered empty in the projected state.)

- All other variables of $\mathcal{N}$ have the same values in state $Proj(s)$ as in state s.

We complete the second part of the proof (Lemma 9.15) using the Refinement Mapping theorem and using the mapping function *Proj*. Notice that is not as simple as it might appear. If all actions of $\mathcal{N}^+$ that change the projected state are actions of $\mathcal{N}$, then it would be quite easy. Unfortunately we have a complication. There are actions of $\mathcal{N}^+$ that can cause a local reset. An example is the receipt of a p_{resp} packet with $p_{resp}.mode = reset$. Such actions change the projected state but are not actions of $\mathcal{N}$. However, we will show that such actions cannot appear in executions of $\mathcal{N}^+|Q$. This is because Q is a strong enough predicate to ensure not only that property L holds in the projected state, but also that no local reset actions are enabled.

We continue to let $s|u$ denote the current state of $\mathcal{N}_u^+$ *projected onto the original automaton* N_u. We also use $s|(u, v)$ to denote the state of $C_{u,v}$ *projected onto the original automaton* N. *In other words* $s|(u, v) = Proj(s).Q_{u,v}$.

9.3 Phases

We formally define a phase (see Figure 6) on a link, as an interval during which the checking/correction procedure works on that link. As we have seen, the checking/correction procedure works by setting $phase_u[v] = true$, and attempting to send out a numbered request packet. If a response is received with a matching number then $phase_u[v]$ is set to false. We make this more precise below.

A (u, v) *phase* for execution α is any interval γ of α such that

- $u = l(u, v)$

- γ begins with a state s_i of α such that $s_i.phase_u[v] = false$ and $s_{i+1}.phase_u[v] = true$.

- γ ends with the first state $s_j, j > i$ of α in which $s_j.phase_u[v] = false$.

Notice that a (u, v) phase is only defined for a leader edge (u, v).

The definition of a phase allows the last state of a phase to overlap the first state of the next phase. Clearly we can divide an execution α into consecutive (u, v) phases as well as intervals that lie between (u, v) phases. Thus we can speak of the i-th (u, v) phase in α in this division.

Our first lemma states that in between phases, at most one data packet is sent on $C_{u,v}$. Thus we can think of an execution (from the point of view of any link $C_{u,v}$) as alternating between a phase during which requests are sent and responses are received, followed by a period where at most one data packet is sent on $C_{u,v}$.

Lemma 9.4 *Between two consecutive (u, v) phases on link (u, v) at most one $\text{SEND}_{u,v}(p_{data})$ event can occur.*

Proof: The first $\text{SEND}_{u,v}(p_{data})$ event after the i-th (u, v) phase will set $phase_u[v] = true$ which begins the $i + 1$-st phase. ▮

The proofs of the next two lemmas are quite tedious and can be found in [Var93]. The first lemma states that a single phase completes in constant time. Intuitively, the time taken to complete a phase consists of the time to start a phase (which may involve the sending of a data packet) followed by the time to send a request and receive a matching response. The second lemma shows that data packets are sent out on a link in a bounded time after they are placed on the queue for the link. Intuitively, this is because a phase takes constant time to complete and if the data queue is non-empty, the code sends at least one data packet between consecutive phases.

Before we state these lemmas, we define a quantity t_p. Intuitively, t_p is the the time it takes to complete a phase. We let $t_p = 6t_n + 12t_l$.

The major components of the time to complete a phase are sketched in Figure 12. Since the free variables may be incorrect in the initial state, the first packet sent on a link can be dropped. However, in constant time, the links stop dropping packets. Then, after the possible sending of data packet, a request is sent out by u in constant time. Next, after the possible sending of a data packet, a response is sent by v in constant time.

Lemma 9.5 Phase Rate: *For all α and any leader edge (u, v) and any integer x, at least x (u, v) phases will have completed within $(x + 1) \cdot t_p$ time units after the start of α.*

Proof: Details can be found in [Var93]. The reason why we need $(x + 1) \cdot t_p$ time (instead of $x \cdot t_p$ time) to complete x phases is as follows. Suppose in the initial state of α, $phase_u[v] = true$; it may take t_p time before $phase_u[v]$ becomes *false*. Thus, the first (u, v) phase in α may be a "partial phase". However, the definition of a phase does not allow us to consider a "partial phase" as a phase. ▮

The following lemma shows that a data packet is sent within a phase time.

Lemma 9.6 Data Packet Rate: *For any $\alpha = s_0, a_1, \ldots$ and any (u, v), if $s_0.|queue_u[v]| > 0$ then a $\text{SEND}_{u,v}(p_{data})$ occurs within t_p time units after s_0.*

Proof: See [Var93] ▮

9.4 Clean Phases

We know that a link can drop packets if a packet is sent when the link already has a packet. To be sure that packets will not be dropped, the sender must never think that a link is free when it isn't. In this case, we say the link is "drop-free".

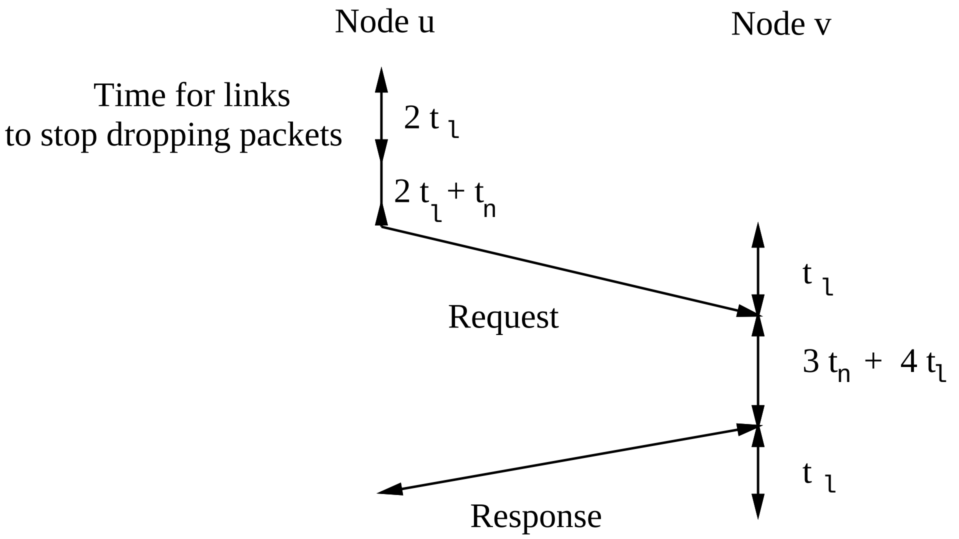

Figure 12: The major components of the time required to complete a phase.

Let $F_{u,v}$ denote the predicate of $\mathcal{N}^+$ defined by: $(\mathit{freeq}_u[v] = \mathit{true}) \rightarrow (Q_{u,v} = \mathit{nil})$. We also say that (u, v) is *drop-free* in state s of $\mathcal{N}^+$ if $s \in F_{u,v}$.

It is easy to show that once a link is drop-free, it remains drop-free. Intuitively, this is because the sender will not record the link as free unless it receives a free signal from the link, which means that there is no packet on the link. It is also easy to show that a link (u, v) becomes drop-free in constant time – in fact, after the first packet sent on the link. This follows because after the first packet is sent on the link, the sender records the link as being busy, and this trivially satisfies the drop-free predicate.

Before studying the structure of phases, we introduce some notation to denote the set of counter values in a (u, v) subsystem. These include counters in any request or response packets, and the counter stored at node v. These variables are sketched in Figure 13.

For any state s we define the derived variables $\mathit{reqcount}(u, v)$, $\mathit{respcount}(u, v)$ and $\mathit{countset}(u, v)$ as follows:

1. If $Q_{u,v} = p_{req}$ then $\mathit{reqcount}(u, v) = p_{req}.\mathit{count}$; otherwise $\mathit{reqcount}(u, v) = \mathit{undefined}$
2. If $Q_{v,u} = p_{resp}$ then $\mathit{respcount}(u, v) = p_{resp}.\mathit{count}$; otherwise $\mathit{respcount}(u, v) = \mathit{undefined}$.
3. $\mathit{countset}(u, v)$ is the set formed by the union of the values in $\mathit{respcount}(u, v)$, $\mathit{reqcount}(u, v)$ and $\mathit{count}_v[u]$

In order for a (u, v) phase to work correctly, we need the phase to follow the structure describe in Figure 6. For this to happen, when a numbered request is first sent during a phase, the number of the request should not already present at the receiver or in the channels. If this is not the case, it is possible for an incorrect response to be accepted in the phase. We formalize this notion of a link and a phase being "clean" using five conditions. The second condition is the crucial condition; the other four are supporting conditions required to ensure that the clean predicate is closed.

We say that a leader edge (u, v) is *clean* in state s of $\mathcal{N}^+$ iff all the following predicates are true in s:

1. (v, u) and (u, v) are drop-free.
2. If $\mathit{phase}_u[v] = \mathit{false}$ then $\mathit{count}_u[v] \notin \mathit{countset}(u, v)$

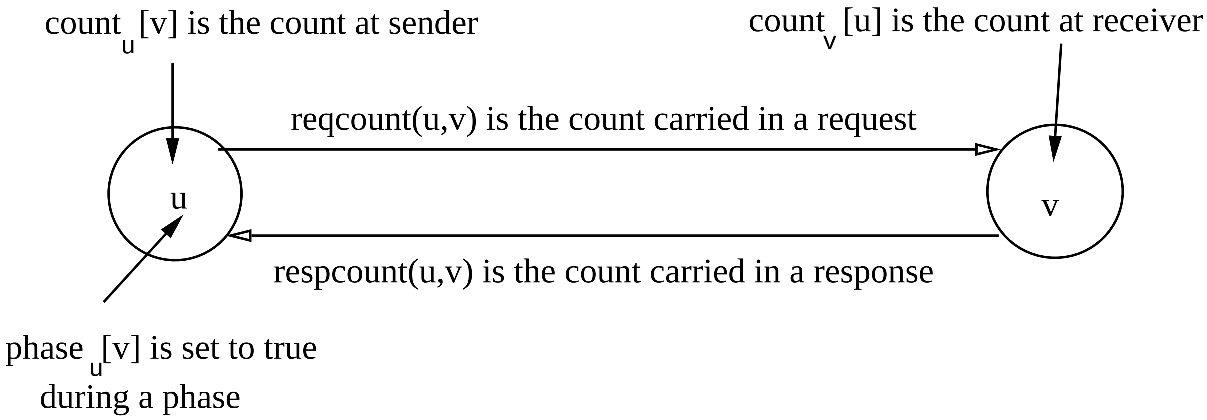

Figure 13: The key variables used in the definition of a clean link: *countset*(u, v) is the union of the count at the receiver and any count values in request and response packets. A clean link ensures that between phases, the sender count value is not in *countset*.

3. If $phase_u[v] = true$ then $reqcount(u, v) = count_u[v]$ or $reqcount(u, v) = undefined$
4. If $phase_u[v] = true$ and $respcount(u, v) = count_u[v]$ then $respcount(u, v) = count_v[u]$.
5. If $count_v[u] = count_u[v]$ then $Q_{u,v} \notin P_{data}$.

The first condition ensures that the links in both directions are drop-free. The second condition ensures that when a numbered request is first sent during a phase, the number of the request is not already present at the receiver or in the channels. This is the important property of a clean link.

The third condition states that during a phase, any requests must carry the sender's number. The fourth condition states that during a phase if there is a matching response on the channel from the receiver to the sender, then the receiver has the same number as the sender. The fifth condition says that if during a phase the sender and receiver numbers are the same, then there can be no data packet in transit from the sender to the receiver. The fifth condition (as we will see below) follows from the fact that the sender will not send a data packet during a phase.

A (u, v) phase p is clean if (u, v) is clean in the first state of p.

Intuitively, a clean phase will contain an action to send a request packet at, u followed by the receipt of this packet at v followed by the sending of a response by u followed by the receipt of the response at u. The receipt of the matching response ends the phase. This is shown in Figure 6. Thus a clean phase will ensure that the response received at node u will correspond to the requesting information at node u.

A nice property is that once an edge becomes clean, it remains clean.

Lemma 9.7 *For any transition $(s, \pi, \tilde{s})$ and any leader edge (u, v), if (u, v) is clean in s then (u, v) is clean in $\tilde{s}$.*

Proof: See [Var93] for details. However, it is not hard to see informally see why this is true. First, as we have seen before, once a link is drop-free, it remains drop-free. Next, we know from the fourth condition that a matching response can only be received if the receiver has the same number as the sender. Also from the second condition, any requests present during a phase must have the same number as the sender. Now the sender always increments its number after receiving a matching response. Thus after a phase is over, the number of the sender must be different from any of the numbers present in the receiver or channels: this is the second condition. The third condition follows from the fact that a phase begins by sending a request with the sender's number, and subsequent retransmissions of requests carry the same number. Next, if a matching response is present, it must have been sent by the receiver during the phase. But, by the third condition, the receiver cannot change its number after it sent the response. Thus if there is a matching response, the number of the receiver must be the same as the sender, which is the fourth condition.

The fifth condition follows because the sender never sends a data packet during a phase. By the second condition, at the start of the phase the receiver number is not equal to the sender number; thus the two numbers become the same only after the receiver has received a request. But the receipt of this request "flushes" out any data packets that were in transit from sender to receiver at the start of the phase. This (taken together with the fact that the sender never sends a data packet during a phase) is what makes the fifth condition hold. ∎

The following lemma explains why the request-response matching procedure is stabilizing.

Lemma 9.8 *A leader edge (u, v) is clean in all states after the fifth (u, v) phase in any execution α.*

Proof: (Idea) We first show that after at most two phases (v, u) and (u, v) are both drop-free. This follows since some packet (i.e., at least one request and at least one response) must have been sent on either link by this time. Next consider the end of the second phase. In this state, there must be some $c \in \{0, \ldots, 3\}$ such that $c \notin \mathit{countset}(u, v)$. This follows because, by definition, $\mathit{countset}(u, v)$ has a maximum of three elements. Now consider the first time after the end of the second phase that $\mathit{count}_u[v] = c$. This must occur at the end of a phase because $\mathit{count}_u[v]$ is only incremented at the end of a phase. Also this must occur at or before the end of the fifth phase because $\mathit{count}_u[v]$ increases by 1 mod 4 at the end of each phase.

It is easy to see that when $\mathit{count}_u[v]$ first becomes equal to c, $c \notin \mathit{countset}(u, v)$. Thus at or before the end of the fifth phase, $\mathit{phase}_u[v] = \mathit{false}$ and $\mathit{count}_u[v] \notin \mathit{countset}(u, v)$. Thus at or before the end of the fifth phase, edge (u, v) is clean. Finally, by Claim 9.7, (u, v) is clean in all subsequent states of α. ∎

Now we have reached the goal of this subsection. First we define:

Let C be the predicate of $\mathcal{N}^+$ such that for all leader edges (u, v), (u, v) is clean in all states in C. Recall that $\mathcal{N}^+|C$ is the automaton that is identical to $\mathcal{N}^+$ except that all leader edges are clean in any initial state.

Theorem 9.9 *$\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$ in time $6t_p$.*

Proof: Follows directly from Lemma 9.7, Lemma 9.8 and Lemma 9.5 and the Execution Convergence theorem, Theorem 9.3. Lemma 9.7 shows that each leader edge becomes clean after at most 5 phases which by Lemma 9.5 takes at most $6t_p$ time. Lemma 9.7 shows that once a leader edge is clean it remains clean. Then the Execution Convergence theorem (Theorem 9.3) shows that all leader edges become clean in at most $6t_p$ time. ∎

9.5 Quiet Links: Establishing Link Predicates

Clean phases are only useful because they allow local predicates to be established as we show in this section. We know from the last section that every execution has a suffix that is clean. When we say that $L_{u,v}$ holds in a state s of $\mathcal{N}^+$, we mean that $(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v}$.

Recall that $<$ is the partial order on edges associated with reset function f. Consider a link (u, v) such that there is no other link $\{w, x\} < \{u, v\}$. We will show that if in the first state of a clean (u, v) phase $\mathit{mode}_u[v] = \mathit{snapshot}$ then at the end of the phase, a true snapshot of the (u, v) subsystem is obtained. In particular, it is possible at the end of such a phase to determine whether $L_{u,v}$ holds at the end of the phase. If it does not hold, $\mathit{mode}_u[v]$ is changed to *reset*. In a similar fashion, we can show that any clean

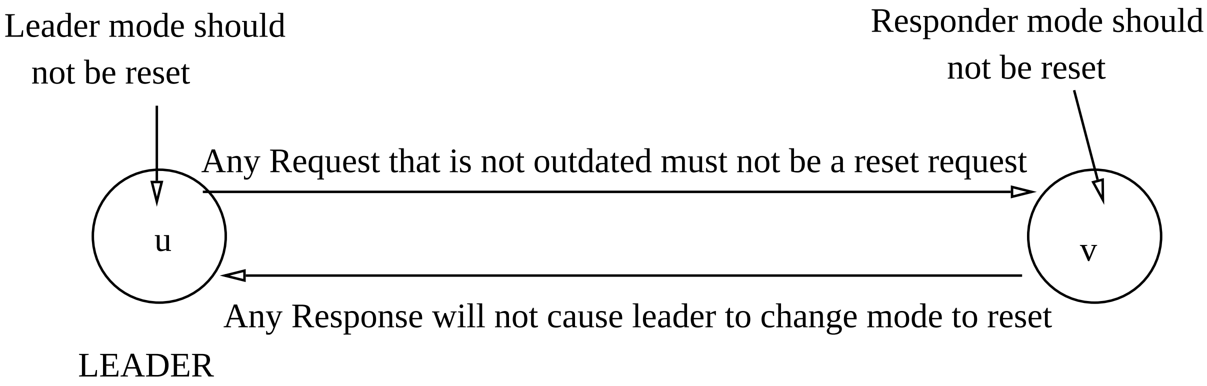

Figure 14: A leader edge is quiet if it clean, its local predicate holds, and it satisfies the conditions sketched in the figure.

phase whose initial state has $mode_u[v] = reset$ will guarantee that $L_{u,v}$ holds at the end of the phase. The net effect is that $L_{u,v}$ holds by end of the second (u,v) phase of a "clean" execution.

However, we want to show not only that $L_{u,v}$ holds by end of the second (u,v) phase but also that no more "reset" actions can occur after this point so that $L_{u,v}$ will remain true. This motivates the following definitions of a quiet link. Please refer to Figure 14 for an intuitive explanation.

We say that a leader edge (u,v) is quiet in state s of $\mathcal{N}^+$ iff the following predicates hold in s:

1. (u,v) is clean.
2. $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$
3. $(mode_u[v] \neq reset)$ and $(mode_v[u] \neq reset)$.
4. If $Q_{u,v} = p_{req}$ and $p_{req}.mode = reset$ then $p_{req}.count = count_v[u]$.
5. If $Q_{v,u} = p_{resp}$ and $p_{resp}.count = count_u[v]$ then $(s|u, nil, nil, p_{resp}.node\_state) \in L_{u,v}$

Notice that the fourth condition above ensures that there is no reset request on the link that could be accepted by the receiver at a later state. The fifth condition ensures that any snapshot information in a matching response will not cause the sender to change its mode to *reset*.

Our goal is to show that eventually all links are quiet.

Let $\mathcal{Q}$ be the predicate set consisting of the following predicate for each leader edge in G. The predicate for each leader edge (u,v) is that (u,v) is quiet. Let Q be the predicate that is the intersection of all predicates in $\mathcal{Q}$.

We will extend the partial order $<$ (which was defined on undirected pairs of nodes) to leader edges by assuming that that leader edge $(u,v) <$ leader edge (w,x) iff $\{u,v\} < \{w,x\}$. Next, we will use the Execution Convergence theorem (Theorem 9.3) to show that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$. We know that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$. Thus it is sufficient to show that $\mathcal{N}^+|C$ is stabilized to Q using predicate set $\mathcal{Q}$ and partial order $<$. To show this, we will first show two lemmas:

First we state the required stability condition: that a leader edge (u,v) will remain quiet as long as all leader edges less than or equal to (u,v) are quiet.

Lemma 9.10 *Consider a leader edge (u,v). If every leader edge $(w,x) \leq (u,v)$ is quiet in some state s of $\mathcal{N}^+|C$, then for any transition $(s,\pi,\tilde{s})$, (u,v) is quiet in state $\tilde{s}$.*

Proof: We omit details. However, the basic idea is simple. We have already seen that the clean predicate is stable. We also know from the definition of local checkability that $L_{u,v}$ is closed for all the actions of the original automaton $\mathcal{N}$. However, in $\mathcal{N}^+$ we added additional actions to send requests and responses. It is easy to verify that the sending of snapshot requests and responses do not affect $L_{u,v}$. However, $L_{u,v}$ is affected by actions that send reset requests and responses on edges to neighbors that are less than (u, v) in the partial order. (If such "reset" actions occur on edges to neighbors that are *not* less than (u, v) in the partial order, then the definition of the local correction function ensures that $L_{u,v}$ remains true.) But such "reset actions" cannot occur on edges less than (u, v) in the partial order, because such edges are quiet by hypothesis.

Next, if $L_{u,v}$ holds, and the snapshot information in matching responses is always correct, the sender will never change its mode to *reset*. But if the sender never changes its mode to *reset*, the sender will never send a reset request. And if the receiver never receives a reset request, the receiver will never change its mode to *reset*. ∎

Next we state the required liveness condition: that a leader edge (u, v) will become quiet in bounded time if all leader edges less than (u, v) are already quiet.

Lemma 9.11 *Consider a leader edge (u, v) and some execution α of $\mathcal{N}^+|C$. If every leader edge $(w, x) < (u, v)$ is quiet in some state s_i of α, then (u, v) is quiet in some state that occurs within $3 \cdot t_p$ of s_i in α.*

Proof: See [Var93] for details. However, the basic idea is simple. Consider the first complete (u, v) phase after s_i. If it is a snapshot phase, and $L_{u,v}$ does not hold at the end of the phase, this will be detected and the next phase will become a reset phase. Thus either the first or second complete phase after s_i will be a reset phase. At the end of the reset phase, (u, v) will become quiet because all the links that (u, v) depends on are already quiet. The upshot is that within two complete phases, (u, v) becomes quiet. But this can take up to $3t_p$ time (where t_p is the time to complete a single phase) because the first phase after s_i can be an incomplete phase. ∎

The last two lemmas immediately give us:

Lemma 9.12 *$\mathcal{N}^+|C$ is stabilized to Q using predicate set $\mathcal{Q}$ and time constant $3t_p$.*

Thus applying the Execution Convergence Theorem (9.3) and using the last lemma, we get:

Lemma 9.13 *$\mathcal{N}^+|C$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time $height(f) \cdot 3t_p$.*

For convenience we define a quantity t_q which intuitively can be thought of as the time after which any execution of $\mathcal{N}^+$ becomes quiet.

We let $t_q = 6t_p + height(f) \cdot 3t_p$; Thus we have the major result of this section:

Lemma 9.14 *$\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time t_q.*

Proof: Follows directly from Lemmas 9.9 and 9.12 and transitivity (Lemma 4.1). ∎

9.6 Projecting Behaviors of $\mathcal{N}|Q$

In this subsection, we prove that if all leader edges are quiet in the initial state of some execution α of $\mathcal{N}^+$, then the external behavior corresponding to α is a behavior of $\mathcal{N}|L$. Intuitively, this is not very hard to believe. This is because for every leader edge (u, v), $L_{u,v}$ holds in every projected state of α. We also know that there will be no reset transitions in α. Formally:

Lemma 9.15 *Every behavior of $\mathcal{N}^+|Q$ is a behavior of $\mathcal{N}(t_p)|L$*

Proof: Follows by using a refinement mapping (Theorem 9.2) using the mapping function *Proj* (see definition in Section 9.2).

First, for any state s of $\mathcal{N}^+|Q$, and any leader edge (u, v), we know (from the definition of Q) that $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$. Thus $Proj(s) \in L$. Thus for any state s of $\mathcal{N}^+$, $Proj(s)$ is a start state of $\mathcal{N}(t_p)|L$

Next consider any transition, $(s, \pi, \tilde{s})$ of $\mathcal{N}^+|Q$. We know from the definition of Q that there are no reset transitions in $\mathcal{N}^+|Q$. Suppose π is a $\textsc{Send}_{u,v}(p_{req})$, $\textsc{Receive}_{u,v}(p_{req})$, $\textsc{Send}_{v,u}(p_{resp})$, or $\textsc{Receive}_{v,u}(p_{resp})$ for any (u, v). Then it is easy to see that $Proj(s) = Proj(\tilde{s})$. But such actions are not actions of $\mathcal{N}$.

If π is a $\textsc{Send}_{u,v}(p_{data})$ action for some edge (u, v), then it must be that p_{data} is at the the head of $s.queue_u[v]$ and $s.free_u[v] = true$. Since $Proj(s).queue_u[v] = s.queue_u[v]$ and $Proj(s).free_u[v] = s.free_u[v]$, the $\textsc{Send}_{u,v}(p_{data})$ event is also enabled in $Proj(s)$. Also, $\tilde{s}.Q_{u,v} = Proj(\tilde{s}).Q_{u,v} = p_{data}$, since (u, v) is drop-free in s. Thus $(Proj(s),\ \textsc{Send}_{u,v}(p_{data})\ ,\ Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$. Similarly if π is a $\textsc{Receive}_{u,v}(p_{data})$ action for some edge (u, v), then it must be that $s.Q_{u,v} = p_{data}$ and $\tilde{s}.Q_{u,v} = nil$. Thus $Proj(s).Q_{u,v} = p_{data}$ and $Proj(\tilde{s}).Q_{u,v} = nil$. and $(Proj(s),\ \textsc{Receive}_{u,v}(p_{data})\ ,\ Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$. Next if π is a $\textsc{Free}_{u,v}$ action for some edge (u, v), then it must be that $s.Q_{u,v} = nil$ and $\tilde{s}.Q_{u,v} = nil$. Thus $Proj(s).Q_{u,v} = nil$ and $Proj(\tilde{s}).Q_{u,v} = nil$ and $(Proj(s), \textsc{Free}_{u,v}, Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$.

Finally suppose that π is any other action of $\mathcal{N}(t_p)|L$. (For example, this category would include internal actions of $\mathcal{N}(t_p)|L$.) Then such actions remain unmodified and the values of all node variables are identical in s and $Proj(s)$. Thus $(Proj(s), \pi, Proj(\tilde{s}))$ is a transition of $\mathcal{N}|L(t_p)$.

Finally consider the timing properties. Suppose a $\textsc{Send}_{u,v}(p_{data})$ event is enabled in $Proj(s)$ for any edge (u, v). Then $s.|queue_u[v]| > 0$. Thus by Lemma 9.6, a $\textsc{Send}_{u,v}(p_{data})$ event occurs within t_p time after s. Suppose a $\textsc{Receive}_{u,v}(p_{data})$ event is enabled in $Proj(s)$ for any edge (u, v). Then $s.Q_{u,v} = p_{data} \neq nil$. From the timing properties of $\mathcal{N}^+|Q$ we know that a $\textsc{Receive}_{u,v}(p_{data})$ will occur in t_l time, and $t_l \leq t_p$.

Consider a $\textsc{Free}_{u,v}$ action for any edge (u, v). Within t_l time units after s, either a $\textsc{Free}_{u,v}$ action will occur or there is a state s' such that $s'.Q_{u,v} \neq nil$. Consider the second case. It is easy to verify from the definition of a UDL that a packet is delivered in time t_l; more precisely, within t_l time units after s', there is a state s'' such that $s''.Q_{u,v} = nil$. Since (u, v) is drop-free in all states of α, $s''.freeq_u[v] = false$. Thus $Q_{u,v} = nil$ will remain true until a $\textsc{Free}_{u,v}$ action occurs within time t_l after s''. Thus a $\textsc{Free}_{u,v}$ action will occur within $3t_l$ time units after state s. In particular, this means that a $\textsc{Free}_{u,v}$ action will occur within t_p time units of any state s such that $\textsc{Free}_{u,v}$ is enabled in $Proj(s)$.

Finally suppose that π is any other locally controlled action of $\mathcal{N}(t_p)|L$. Thus π is an internal action of $\mathcal{N}(t_p)|L$. Notice that π is in the same class say c (with upper bound equal to t_c) in both $\mathcal{N}^+|Q$ and $\mathcal{N}(t_p)|L$. Suppose that π is enabled in $Proj(s)$. Then π is enabled in s. Then in t_c time of s in α either some action in class c occurs or there is some state $\tilde{s}$ such that no action in class c is enabled in $Proj(\tilde{s})$. This follows because if no action in class c is enabled in $\tilde{s}$, then no action in class c is enabled in $Proj(\tilde{s})$, and vice versa. ∎

9.7 Tying up the Proof

We now return to the proof of the Local Correction Theorem, Theorem 6.1. It follows from Lemma 9.15 and Lemma 9.14 and Theorem 9.1.

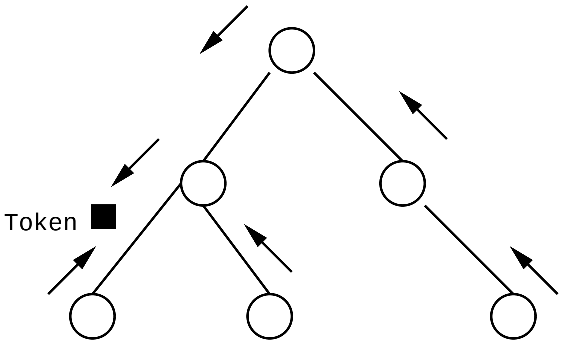

Figure 15: Adding a pointer to each node makes token passing on a tree locally checkable.

10 Simple Example of Local Correction: Token Passing

We describe a complete example application of our local correction theorem. Our local correction method and theorem has been used to construct stabilizing solutions to a number of non-trivial applications including a network reset protocol[APV91, Var93], a synchronizer [AV91], and a stabilizing solution to the end-to-end problem [APV96]. We refer the reader to those papers for these applications. In this section, we give a complete example of the Local Correction theorem to a simple application: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. Our protocol is simple and 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.

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 modeling 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 15.

10.1 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 16.

We assume that there is a function $neighborSet(u)$ that lists the set of neighboring nodes of node u. (Recall that we allowed node automata to depend on the set of neighboring nodes and the leader function.) We assume there is some fixed circular ordering on $neighborSet(u)$ 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:

$\text{ENQUEUE}_{u,v}$ (*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*)

$\text{SEND}_{u,v}(token)$ (*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*)

$\text{FREE}_{u,v}$ (*link to v says it is free, input action*)
  Effect:
    $free_u[v] = true$

$\text{RECEIVE}_{v,u}(token)$ (*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 16: Actions for a node u with respect to a neighbor v in the stabilizing token passing protocol.

- $pointer_u \in neighborSet(u) \cup nil$ (*pointer to token*)
- $last_u \in neighborSet(u)$ (*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 enqueuing 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.

10.2 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 $queue_u[v]$, $Q_{u,v}$, $queue_v[u]$, 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.

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 10.1 *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 $\text{ENQUEUE}_{u,x}$ 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 $\text{SEND}_{u,x}(token)$ 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 $\text{FREE}_{u,x}$ event. Then this action does not change the concerned variables and hence $\tilde{s}$ satisfies $L_{u,v}$.

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 $L_{u,v}$. ∎

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

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 17.

Lemma 10.2 *The function f defined in Fig. 17 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 the definition of a reset function in Section 5.3.

- **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 17 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 17 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 10.3 *The network automaton $\mathcal{T}$ is locally correctable to L using link predicate set $\mathcal{L}$ and the reset function f defined in Fig. 17.*

By applying the Local Correction theorem (Theorem 6.1) (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 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.

11 Other Theorems on Local Checking

We might speculate 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 (stated in [Var93] 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.

Another complementary result is the *Global Correction* theorem stated in [APVD94, Var93]. 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. Formal statements of these theorems can be found in [APVD94, Var93].

12 Implementing Local Checking in Real Networks

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 [APV94]. 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.

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. Periodic checking is sufficient (instead of requiring a snapshot) 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). It appears that the following fact is true: *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. However, it is easy to construct examples of protocols that are not one-way checkable for which this is not true.

The significance of this observation is that the Internet protocol design philosophy ([Cla91]) advocates the use of so-called "soft-state" in which each node periodically sends its state to its neighbors. Soft-state is advocated in [Cla91] as a method of making protocols more robust. However, we can see that soft-state and periodic sending is not a panacea: for example, periodic sending is insufficient if the predicates are not local and if the predicates are not one-way.

13 Conclusions

In this paper, we have formally defined the notions of local checkability and correctability and described a 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 gave an example application of stabilizing a mutual exclusion protocol. Other, more non-trivial applications include stabilizing a network reset protocol[APV91, Var93], a synchronizer [AV91], and a stabilizing solution to the end-to-end problem [APV96].

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 [APV91]. The price for using global correction is that the stabilization time now becomes proportional to the number of nodes. Examples of problems that can be solved by local checking but without local correction can be found in [Var93].

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. More detailed references can be be found in [APV91], [APVD94], [AV91], and [Var93].

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.

[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).

[APV91] 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.

[APV94] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Creating self-stabilizing protocols by using a reset protocol. In *Proc. of the 1994 INFOCOM*, 1994.

[APV96] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilizing end-to-end communication. *Journal of High Speed Networks*, 5(4), October 1996.

[APVD94] B. Awerbuch, B. Patt, G. Varghese, and S. Dolev. Self-stabilization by local checking and network correction. In *Proc of Workshop on Distributed and Graph Algorithms (WDAG), Netherlands*, 1994.

[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.

[BP89] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[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.

[Cla91] David Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings ACM SIGCOMM 88*, 1988.

[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.

[DIM90] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[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.

[GM90] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. Technical Report TR-90-20, Dept. of Computer Science, University of Texas at Austin, June 1990.

[IJ90a] Amos Israel and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[IJ90b] A. Israeli and M. Jalfon. Self-stabilizing ring orientation. In *Proc. 4th Workshop on Distributed Algorithms*, Italy, September 1990.

[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.

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

[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.

[MMT91] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. In *CONCUR 91*, pages 408–423, 1991.

[MP91] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83, 1991.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Lang. and Syst.*, 4(3):455–495, 1982.

[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.

[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.

[Var93] George Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.

[Var94] George Varghese. Self-stabilization by counter flushing. In *Proceedings of the 13th PODC*, Los Angeles, California, August 1994.