rem. In essence, this result states that any locally correctable protocol can be stabilized using a simple transformation. The transformation involves the addition of extra actions to do local checking and correction. Section 5.4 also contains a formal description of the transformation. The next two sections contains a proof of the Local Correction Theorem. We first provide an intuitive "proof" that presents the main ideas and then present a formal proof. The formal proof is important because it shows how we use the proof techniques of Chapter 4 to formally prove stabilization results. The proofs of the Tree and Global Correction theorems in later chapters are much more intuitive; the formal proof in this chapter provides an important example of how such intuitive proofs can be formalized.

Finally, Section 5.7 argues that the method of local checking and correction (that is formalized by the Local Correction Theorem) is practical, and can be added to real networks without an appreciable loss in efficiency.

5.1 Modelling Network Protocols

For the rest of this thesis, we will restrict ourselves to proving stabilization properties for network protocols and network automata. To model a network protocol, we need to model the network topology, the links between nodes, and the nodes themselves. Our model is essentially the standard asynchronous message passing model except for three differences:

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

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

5.1.1 Modelling Network Topology

We use a directed graph G to specify the network topology. For any two neighboring nodes u and v, there are two edges (u,v) and (v,u). The network automaton will have a undirectional channel corresponding to each directed edge.

In addition, we require that G satisfies the following property: there is a function that for any pair of neighboring nodes u and v in G assigns one of the two nodes as the "leader" of the edges (u,v) and (v,u). Thus we are really requiring a way to break symmetry between neighboring nodes in G.

It is possible to remove this assumption (of having a leader function) at the cost of some increased complexity in the protocols. However this assumption is not restrictive in practice. In a real implementation, if every node has a unique ID then a simple stabilizing protocol can elect the minimum ID node as leader. Each node can periodically transmit its ID to its neighbor and both nodes choose the minimum ID. If the nodes do not have IDs then an equally simple randomized protocol can elect a leader on every link. We prefer to encode the leader function directly in the graph G instead of presenting these simple protocols explicitly.

In summary, the nodes and edges of G correspond to the actual physical topology of the network, while the leader function describes a way to break symmetry between neighboring nodes. Formally:

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

Definition 5.1.1 *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.

The following definition of a leader edge is useful later. Of the two possible edges between two neighbors, it produces the edge directed away from the leader.

Definition 5.1.2 *We call* (u,v) *a* leader edge *of* G *if* $(u,v) \in E$ *and* $l(u,v) = u$.

5.1.2 Modelling Network 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 [DIM91a] that almost any non-trivial task is impossible in such a setting. Thus in a stabilizing setting it is necessary to define Data Links that have bounded storage.

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

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

Definition 5.1.3 *We say that $C_{u,v}$ is the UDL corresponding to ordered pair (u,v) and with link delay t if $C_{u,v}$ is the UIOA defined in Figure 5.1.*

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

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

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 with an upper bound called the *link delay* which is equal to t for both classes.

Figure 5.1: Unit Storage Data Link automaton

5.1.3 Modelling Network Nodes

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.

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.

Definition 5.1.4 *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* $\text{SEND}_{u,v}(p)$ *and input actions* $\text{RECEIVE}_{v,u}(p)$ *and* $\text{FREE}_{u,v}$ *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}.*[1]
- *Actions of N other than* $\text{SEND}_{u,v}(p)$ *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* $\text{SEND}_{u,v}(p)$ *and the input action* $\text{FREE}_{u,v}$ *at node N is as shown in Figure 5.2.*
- *Each* $\text{SEND}_{u,v}(p)$ *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 $\text{SEND}_{u,v}(p)$ action at a node is a locally controlled action. Also, in all the network automata described in this thesis, the transitions in automaton N_u will only depend on the 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.

[1]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.

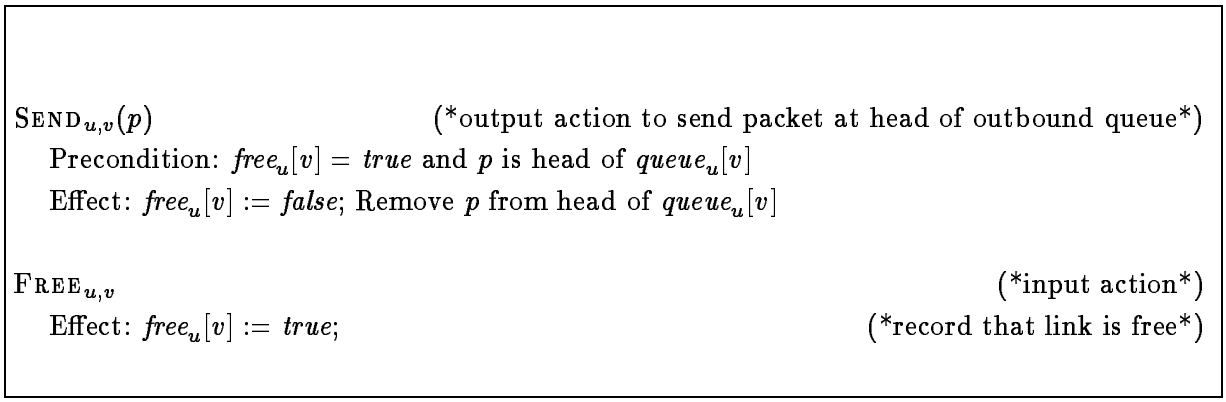

Figure 5.2: Code at a node automaton to send a data packet and to respond to a free signal from a link

5.1.4 Network Automata

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

Definition 5.1.5 *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.

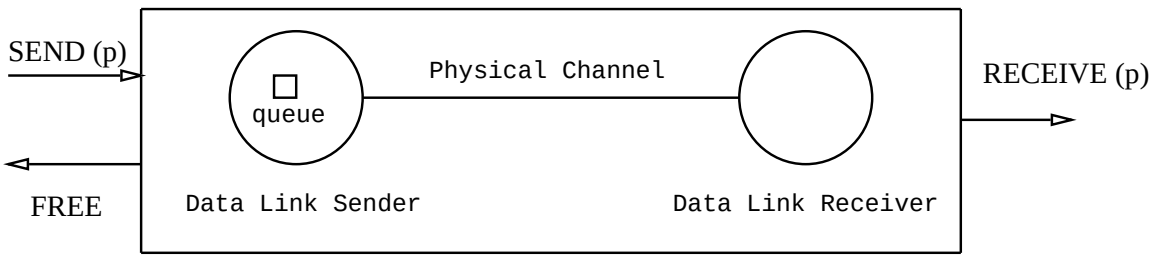

Figure 5.3: Implementing a UDL over a physical channel

5.2 Implementing Our Model in Real Networks

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

Figure 5.3 shows the structure of such a Data Link protocol over a physical link. The sender end of the Data Link protocol has a queue that can contain a single packet. When the queue is empty, the FREE signal is enabled. When a SEND(p) arrives and the queue is empty, p is placed on the queue; if the queue is full, p is dropped. If there is a packet on the queue, the sender end constantly attempts to send the packet. When the receiving end of the Data Link receives a packet, the receiver sends an ack to the sender. When the sender receives an ack for the packet currently in the queue, the sender removes the packet from the queue.

If the physical channel is initially empty and the physical channel is FIFO (i.e., does not permute the order of packets), then a standard stop and wait or alternating bit protocol [BSW69] will implement a UDL. However, if the physical channel can initially store packets, then the alternating bit protocol is not stabilizing [Spi88a]. There are two approaches to creating a stabilizing stop and wait protocol. Suppose the physical channel can store at most X packets in both directions. Then [AB89]

suggest numbering packets using a counter that has at least $X + 1$ values. Suppose instead that no packet can remain on the physical channel for more than a bounded amount of time. [Spi88a] exploits such a time bound to build a stabilizing Data Link protocol. The main idea is to use either numbered packets or timers to "flush" the physical channel of stale packets.

A stop and wait protocol is not very efficient over physical channels that have a high transmission speed and/or high latency. It is easy to generalize a UDL to a *Bounded Storage Data Link* or BDL that can store more than one packet. For instance, the FREE signal for a BDL should be modified to include the number of packets currently stored in the BDL. It is also easy to implement a BDL over a physical channel with either bounded storage or bounded delay using the techniques described in [AB89] and [Spi88a]. We prefer to use a UDL for the rest of this thesis as it provides a simple and elegant interface. However, the reader concerned about efficiency should be aware that all the protocols in this thesis can be modified (slightly) to work with BDLs.

Finally, there is one last concern about UDLs. We have seen that real implementations will use Data Links that *stabilize* to a UDL. However, in our model every link is assumed to actually be a UDL. Let S be a network in which each link *stabilizes* to the behaviors of a UDL and such that every node automaton is a UIOA. Let $\tilde{S}$ be the same network except that every link is replaced by a UDL. Now a UDL is a UIOA and hence is suffix-closed. Thus a nice consequence of Theorem 3.5.7 is that if $\tilde{S}$ stabilizes to the behaviors in some problem P, then so does S. Thus, to prove a stabilization result about S it suffices to prove the stabilization result about $\tilde{S}$. This is an example of why the modularity theorem (Theorem 3.5.7) is important.

5.3 Locality

In this section, we reconsider the definitions of local checkability that we introduced in Chapter 4. Our new definitions will be slightly more complex because of the presence of channels between nodes. *These new definitions will be used for the rest of the thesis.*

5.3.1 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'. For now we will concentrate on *link subsystems* that consist of a pair of neighboring nodes u and v and the channels between them. In Chapter 10, we will discuss how our methods can be generalized to arbitrary subsystems.

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

Definition 5.3.1 *We define the* (u, v) *link subsystem of* $\mathcal{N}$ *as the composition of* N_u, $C_{u,v}$, $C_{v,u}$, *and* N_v.

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

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

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

Definition 5.3.2 *We say that a state* s *of* $\mathcal{N}$ satisfies *a local predicate* $L_{u,v}$ *of* $\mathcal{N}$ *iff* $(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v}$.

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

Definition 5.3.3 *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. As in Chapter 4, we will require that the conjunction of the local properties is non-trivial – i.e., there is some global state that satisfies all the local properties.

Definition 5.3.4 *$\mathcal{L}$ is a* link predicate set *for $\mathcal{N} = Net(G, N)$ if for each $(u, v) \in G$ there is some $L_{u,v}$ such that:*

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

Definition 5.3.5 *The conjunction of a link predicate set $\mathcal{L}$ is the predicate $\{s : s$ satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}\}$. We use $Conj(\mathcal{L})$ to denote the conjunction of $\mathcal{L}$.*

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

5.3.2 Local Checkability

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

Definition 5.3.6 *A network automaton $\mathcal{N}$ is locally checkable for predicate L using link predicate set $\mathcal{L}$ if:*

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

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

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

5.3.3 Local Correctability

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

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

We will insist that f meet two requirements so that f can be used for local correction (Definition 5.3.7).

Assume that the property L holds if a local property $L_{u,v}$ holds for every edge (u,v). The first requirement is that if any (u,v) subsystem does not satisfy $L_{u,v}$, then

applying f to both u and v should result in making $L_{u,v}$ hold. More precisely, let us assume that by some magic we have the ability to simultaneously:

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

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

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

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

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

Definition 5.3.7 *We say f is a local reset function for network automaton $\mathcal{N} = Net(G, N)$ with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ and partial order $<$, if for any state s of $\mathcal{N}$ and any edge (u, v) of G:*

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

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

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

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

Definition 5.3.8 *A network automaton $\mathcal{N}$ is* locally correctable *to L using link predicate set $\mathcal{L}$, local reset function f, and partial order $<$ if:*

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

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

Definition 5.3.9 *For any partial order $<$, height$(<)$ is the length of the maximum length chain in $<$.*

5.4 Local Correction Theorem

5.4.1 Overview

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.[2]

5.4.2 Precise Statement of the Result

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.

Definition 5.4.1 *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.

[2] 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.

Definition 5.4.2 *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 5.4.3 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 $\tilde{c} \cdot height(<)$.*

5.4.3 Overview of the Transformation Code

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, 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 5.4.

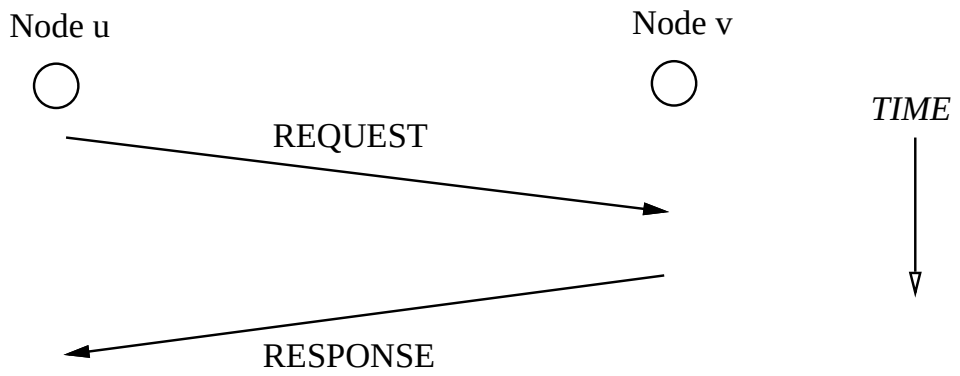

Figure 5.4: The structure of a single phase of the local snapshot/reset protocol

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 $\mathcal{N}^+$. To make the snapshot and reset protocols stabilizing, 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 to $count_u[v]$.

Clearly the *count* values can be arbitrary in the initial state and the first few phases may not work correctly. However, numbering and a few easy checks ensure that in constant time a response will be properly matched to the correct request. Because the links are unit storage, we will see that a space of 4 numbers is sufficient. Our use of numbering is taken from the stabilizing global snapshot protocol of [KP90]. However, our protocol is simpler and more efficient because we are restricted to a single link subsystem.

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.

An irritating issue that we have to deal with in creating $\mathcal{N}^+$ is the issue of scheduling packets to be sent on links. Notice that the checking and correction protocols are going on concurrently with the protocol corresponding to $\mathcal{N}$. To make Theorem 5.4.3 work, we need to ensure that any data packets that are placed on the queue for channel $C_{u,v}$ are sent in constant time. On the other hand, the checking process also needs to send request and response packets that are encoded as control packets.

We build a simple stabilizing scheduler that ensures fair access to the link for each of three packet types: requests, responses and data packets. First we notice that at the leader end of a link, only requests and data packets need to be sent. At the other end, only responses and data packets need to be sent.

Consider the leader end of a link first. Suppose $l(u,v) = u$. We know that data

packets should never be sent while a snapshot or reset phase is in progress. Now, we have a variable $phase_u[v]$ at u which is set to true whenever a phase is in progress. The phase ends when a matching response is received and $phase_u[v]$ is set to false. At the end of a phase, the oldest data packet is sent, and a new phase is begun after the data packet is sent. The scheduler is stabilizing because if there is no data packet waiting to be sent, we allow a new phase to begin immediately after the previous phase ends. The net effect is that if there are data packets waiting, we send one data packet between consecutive checking/correction phases. If a problem is discovered during a snapshot phase, we do not do a reset until the next phase, after sending any waiting data packet.

Now consider the end of a link that is not the leader. Suppose $l(u,v) = v$. To give "fair turns" to the response packets we use a variable $turn_u[v]$ that has only two values *data* (for data packets), and *response* (which is the value of *turn* for response packets). No packet can be sent until either its turn arrives or there is no packet of the other type. After a packet of a particular type is sent, the turn is "toggled" to the other type.

5.4.4 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:.
  - $count_u[v] \in \{0 \ldots 3\}$ (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 $\text{SEND}_{u,v}(p)$ (for data packets p) and $\text{FREE}_{u,v}$ actions of the original automaton N_u as shown in Figure 5.5. 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

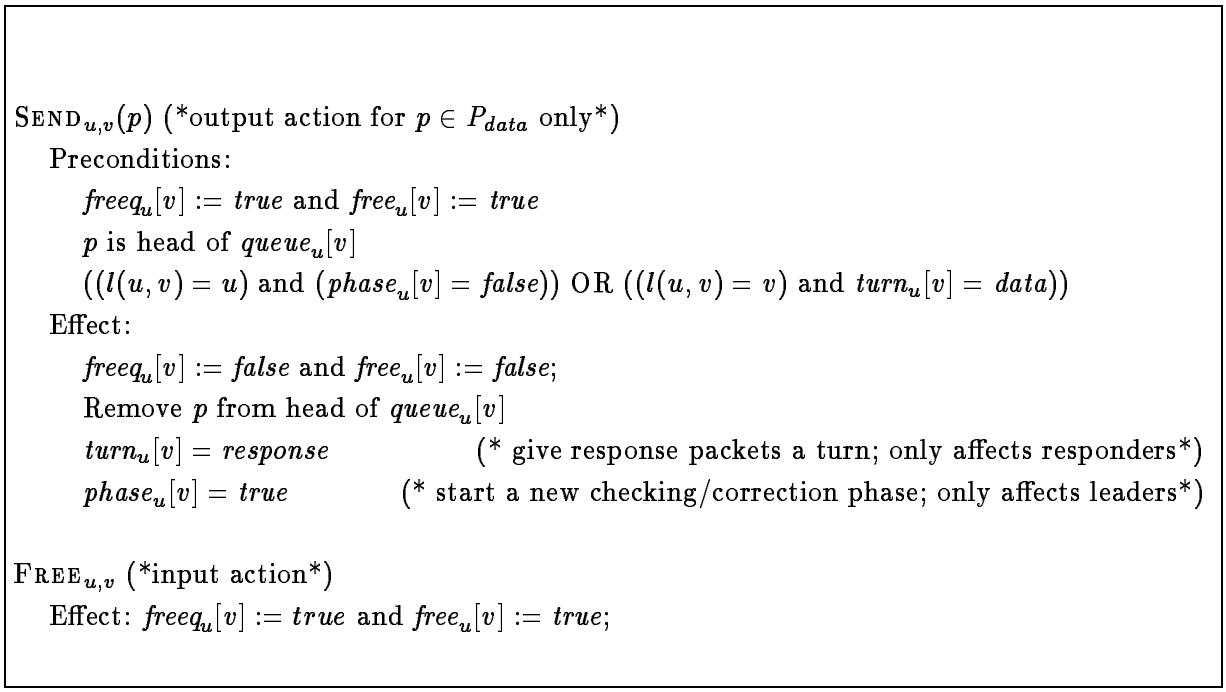

Figure 5.5: Code for the modified SEND$_{u,v}(p)$ actions at a modified node N_u^+.

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 5.6 and 5.7 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^+.

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

$\text{RECEIVE}_{v,u}(p_{req})$ (*input action, receive request at u from v*)
Effect:
If $p_{req}.count \neq count_u[v]$ 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 5.6: Code to send and receive request packets at node 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.

- 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 $\mathcal{N}$.

$\textsc{Send}_{u,v}(p_{resp})$ (*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*)

$\textsc{Receive}_{v,u}(p_{resp})$ (*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 $(s|u, nil, nil, p_{resp}.node_state) \notin L_{u,v}$ 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*)

$count_u[v] := (count_u[v] + 1) \bmod 4$;

Figure 5.7: Code to send and receive response packets at node u.

5.5 Intuitive Proof of Local Correction Theorem

In the previous section we 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 $\mathcal{N}^+$ (described in the last section) is a behavior of $\mathcal{N}$ in which all local predicates hold.

The basic intuition behind the proof is sketched in Figure 5.8 and Figure 5.9. Consider some (u, v) subsystem in which u is the leader. We describe the intuition behind the use of a counter to ensure proper request-response matching, and the intuition behind the local snapshot and reset procedures.

5.5.1 Intuition Behind Counter Based Matching

Recall that in the augmented automaton, both local snapshots and local responses are implemented using a request-response protocol. As we will see below, 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 5.8 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 5.8.

The code of the augmented automaton ensures that correct matching will occur after at most 5 phases by numbering requests and responses. Thus the code uses a counter in the range 0..3. The significance of the number 3 will be seen below. The sender u keeps a counter $count_u[v]$ to number requests and the receiver v keeps a counter $count_v[u]$ to number responses. Node v accepts a new request numbered c only if $c \neq count_v[u]$. When v accepts this request, v also sets $count_v[u]$ to be equal to c. Finally, node u accepts a response numbered c only if $c = count_u[v]$. When u accepts this response, u also increments $count_u[v]$ mod 4. This is implemented in the code and illustrated in the diagram on the right of Figure 5.8.

In the first two phases, packets sent by u and v may be dropped by the links because the links may have packets stored in the initial state. However, it is easy to see from the properties of a UDL, that after at most two phases, the links in both directions are *drop-free* – i.e., any packets sent from that point on will be delivered. Thus we need

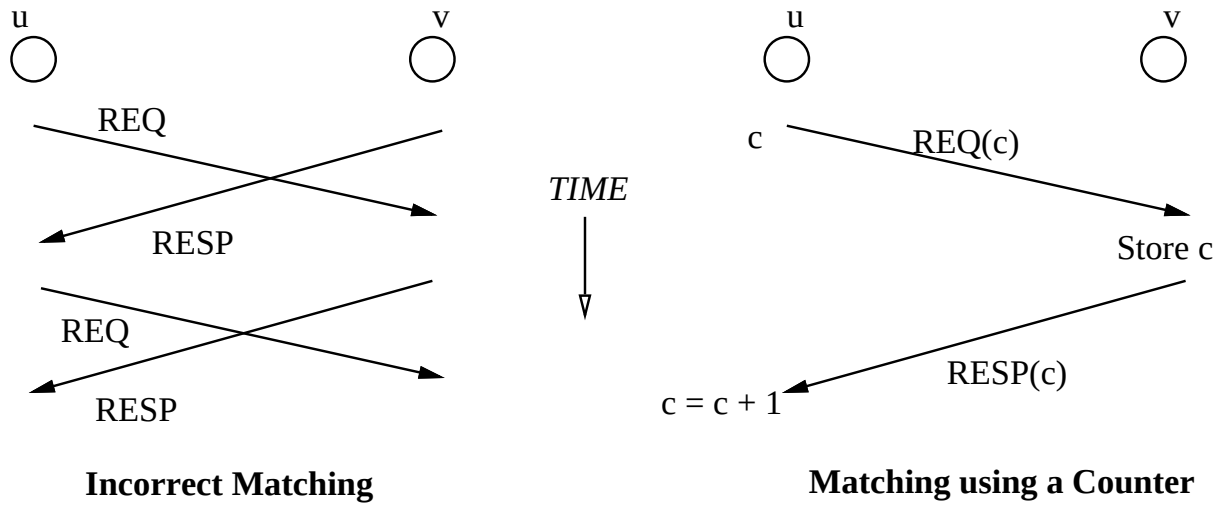

Figure 5.8: Using counter flushing to ensure that request-response matching will work correctly within a small number of phases.

to show that within the next three phases, the request-response matching will begin to work correctly. This follows from a simple paradigm that we call *counter flushing*.

Counter flushing is a general technique that is quite versatile. Chapter 10 describes some more applications of counter flushing. In our case, the counter flushing argument runs as follows:

- There can be at most 3 counter values stored in the two links (i.e., the link from u to v and the link from v to u) and the receiver. This is the significance of the number 3.
- The sender retransmits till it gets a response and so the sender counter will keep being incremented.
- Within 3 increments of the sender counter, the sender counter will reach a "fresh" counter value that is not present in the links and the receiver. This is because the counter space has 4 values, and new counter values can only be created by the sender.
- Suppose the sender sends a request numbered c where c is a fresh value that is not present in the receiver or on the two links. Then when a later response numbered

c is received, this response is a matching response because no aliasing is possible. Also, by the time the sender receives the matching response, the only counter value stored in the two links and the receiver is c. In other words, a freshly numbered request and its matching response will "flush" the (u, v) subsystem of outdated counter values. Hence the term counter *flushing*.

- After all old counter values have been flushed, we say that the (u, v) subsystem is *clean*. It is easy to show that all subsequent phases follow the structure shown in the diagram on the right of Figure 5.8.

5.5.2 Intuition Behind Local Snapshots

The diagram on the left of Figure 5.9 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 5.9.

From the code we know 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 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

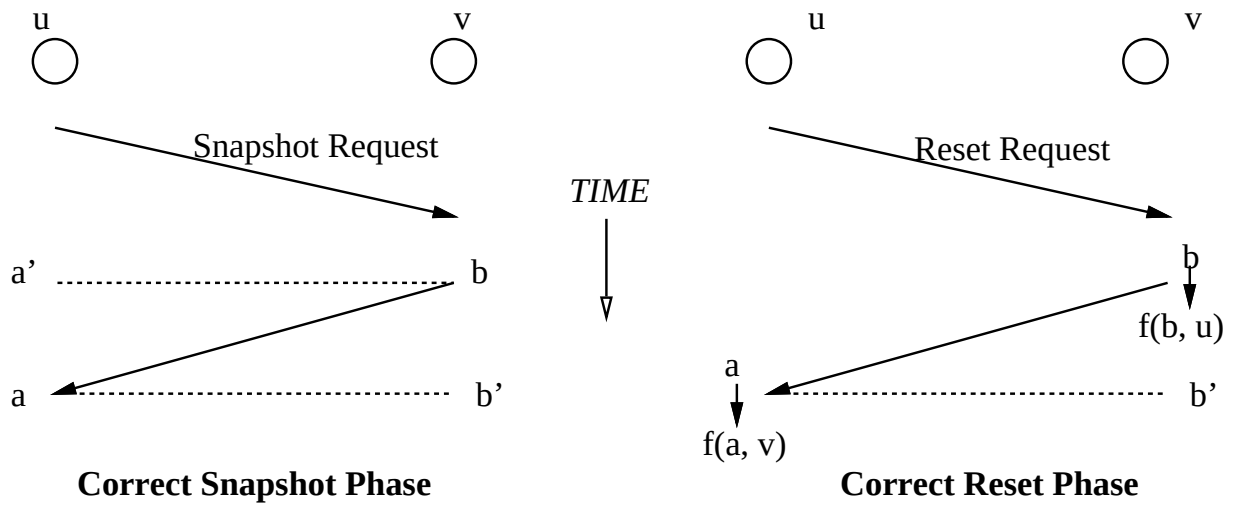

Figure 5.9: Local Snapshots and Resets work correctly if requests and responses are properly matched.

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.

5.5.3 Intuition Behind Local Resets

The diagram on the right of Figure 5.9 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 5.9.

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.

5.5.4 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 $<$.

5.6 Formal Proof of Local Correction Theorem

The reader who is satisfied with the "intuitive proof" given above should skip this section. However, there are a number of details glossed over in the intuitive proof that are spelled out in the formal proof. The formal proof also provides a good example of the proof techniques of Chapter 3. Formal proofs for other major theorems in this thesis like the Tree Correction theorem (Chapter 6) and the Global Correction Theorem (Chapter 8) can be constructed along similar lines.

5.6.1 Overview of Formal Proof

We wish to prove the local correction theorem, Theorem 5.4.3. 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 Chapter 3 in Lemma 3.4.2. 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 3.4.5.
- 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 3.4.3

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

The first part of the proof is described in Sections 5.6.2 to 5.6.4. To show that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$ we use another two step process:

- In Section 5.6.3 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

5.6.3, in Theorem 5.6.14 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 5.6.4 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 5.6.4, in Theorem 5.6.22 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 proof (Section 5.6.5, (Lemma 5.6.23)) 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 3.4.3. 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 5.4.3.

Definition 5.6.1 *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 5.6.23) 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* $s|(u,v) = Proj(s).Q_{u,v}$.

5.6.2 Phases

We formally define a phase (see Figure 5.4) 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.

Definition 5.6.2 *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 5.6.3 *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 are relegated to the appendix. 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.

Definition 5.6.4 *We let $t_p = 6t_n + 12t_l$.*

The major components of the time to complete a phase are sketched in Figure 5.10. 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. The appendix contains more details.

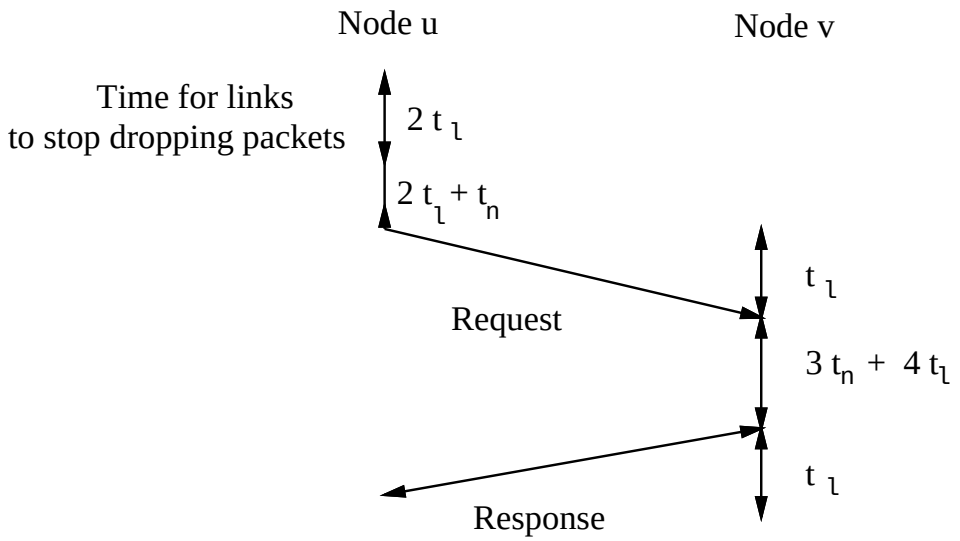

Figure 5.10: The major components of the time required to complete a phase.

Lemma 5.6.5 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: From Lemma B.3.4 and Lemma B.3.5 in Section B.3 of the appendix. 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. ∎

Lemma 5.6.6 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: At the end of Section B.3 in appendix. ∎

5.6.3 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

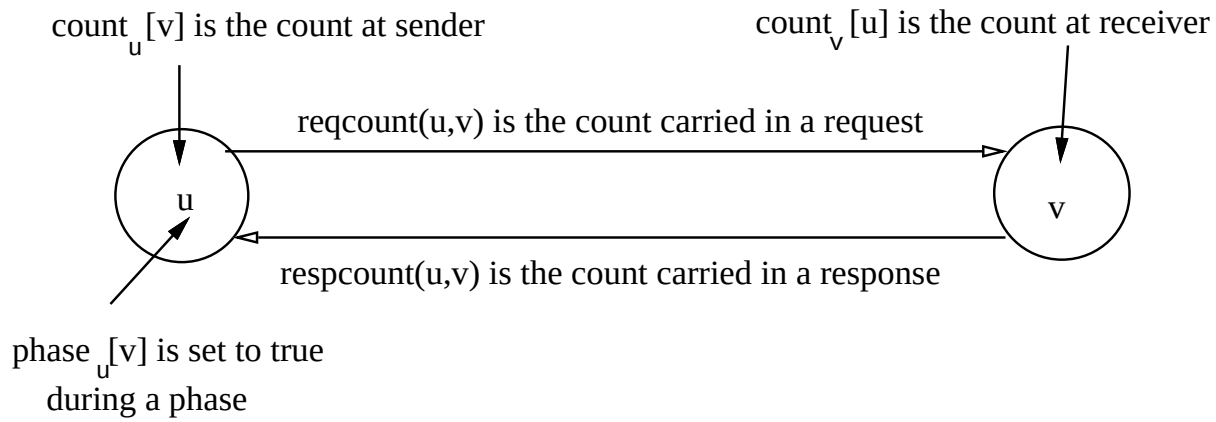

Figure 5.11: The key variables used in the definition of a clean link: $countset(u,v)$ is the union of the count at the receiver and any count values in request and response packets. A clean link ensures that between phases, the sender count value is not in $countset$.

a link is free when it isn't. In this case, we say the link is "drop-free".

Definition 5.6.7 *Let $F_{u,v}$ denote the predicate of $\mathcal{N}^+$ defined by: $(freeq_u[v] = true) \rightarrow (Q_{u,v} = nil)$. We also say that (u,v) is drop-free in state s of $\mathcal{N}^+$ if $s \in F_{u,v}$.*

In the appendix, we 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. In the appendix, we also show 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 5.11.

Definition 5.6.8 *For any state s we define the derived variables $reqcount(u,v)$, $respcount(u,v)$ and $countset(u,v)$ as follows:*

1. *If $Q_{u,v} = p_{req}$ then $reqcount(u,v) = p_{req}.count$; otherwise $reqcount(u,v) = undefined$*

2. *If* $Q_{v,u} = p_{resp}$ *then* $respcount(u,v) = p_{resp}.count$*; otherwise* $respcount(u,v) =$ *undefined.*

3. $countset(u,v)$ *is the set formed by the union of the values in* $respcount(u,v)$, $reqcount(u,v)$ *and* $count_v[u]$

In order for a (u,v) phase to work correctly, we need the phase to follow the structure describe in Figure 5.4. 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.

Definition 5.6.9 *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* $phase_u[v] = false$ *then* $count_u[v] \notin countset(u,v)$

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.

Definition 5.6.10 *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 5.4. 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 5.6.11 *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 Section B.4 in the appendix. 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 5.6.12 *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 (see Claim B.2.2 in the appendix) 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 using Claim B.4.1 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 5.6.11, (u,v) is clean in all subsequent states of α. ∎

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

Definition 5.6.13 *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 5.6.14 *$\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$ in time $6t_p$.*

Proof: Follows directly from Lemma 5.6.11, Lemma 5.6.12 and Lemma 5.6.5 and the Execution Convergence theorem, Theorem 3.4.5. Lemma 5.6.11 shows that each leader edge becomes clean after at most 5 phases which by Lemma 5.6.5 takes at most $6t_p$ time. Lemma 5.6.11 shows that once a leader edge is clean it remains clean. Then the Execution Convergence theorem (Theorem 3.4.5) shows that all leader edges become clean in at most $6t_p$ time. ∎

5.6.4 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 $mode_u[v] = snapshot$ 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, $mode_u[v]$ is changed to *reset*. In a similar fashion, we can show that any clean 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 5.12 for an intuitive explanation.

Definition 5.6.15 *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 $(mode_v[u] \neq reset)$.*

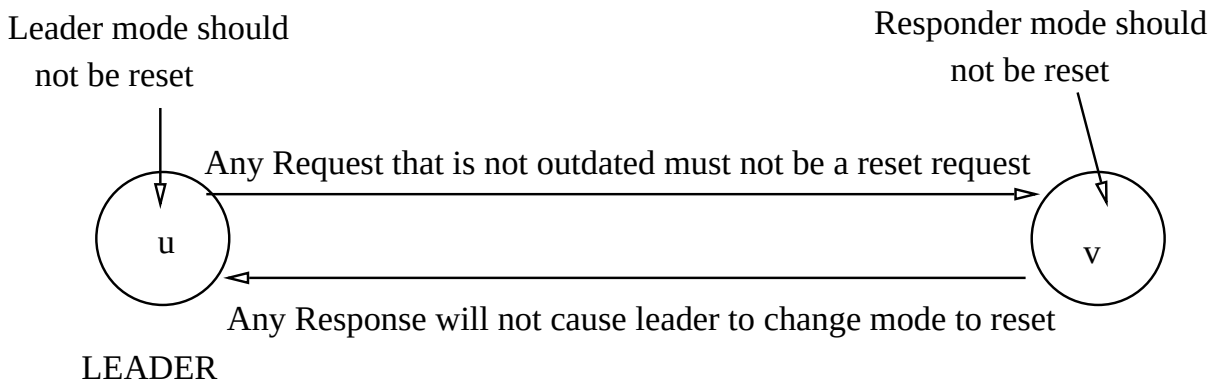

Figure 5.12: A leader edge is quiet if it clean, its local predicate holds, and it satisfies the conditions sketched in the figure.

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.

Definition 5.6.16 *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 3.4.5) 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 5.6.17 *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: In Section B.5 in the appendix. 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 5.6.18 *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: In In Section B.5 in the appendix. 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 5.6.19 *$\mathcal{N}^+|C$ is stabilized to Q using predicate set $\mathcal{Q}$ and time constant $3t_p$.*

Thus applying the Execution Convergence Theorem (3.4.5) and using the last lemma, we get:

Lemma 5.6.20 *$\mathcal{N}^+|C$ 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.

Definition 5.6.21 *We let $t_q = 6t_p + height(f) \cdot 3t_p$;*

Thus we have the major result of this section:

Lemma 5.6.22 *$\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time t_q.*

Proof: Follows directly from Lemmas 5.6.14 and 5.6.19 and transitivity (Lemma 3.1.6). ∎

5.6.5 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 5.6.23 *Every behavior of $\mathcal{N}^+|Q$ is a behavior of $\mathcal{N}(t_p)|L$*

Proof: Follows by using a refinement mapping (Theorem 3.4.3) using the mapping function *Proj* (Definition 5.6.1).

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 $\text{SEND}_{u,v}(p_{req})$, $\text{RECEIVE}_{u,v}(p_{req})$, $\text{SEND}_{v,u}(p_{resp})$, or $\text{RECEIVE}_{v,u}(p_{resp})$ 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 $\text{SEND}_{u,v}(p_{data})$ 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 $\text{SEND}_{u,v}(p_{data})$ 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 $(Proj(s)$, $\text{SEND}_{u,v}(p_{data})$, $Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$. Similarly if π is a $\text{RECEIVE}_{u,v}(p_{data})$ 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 $(Proj(s)$, $\text{RECEIVE}_{u,v}(p_{data})$, $Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$. Next if π is a $\text{FREE}_{u,v}$ 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 $(Proj(s), \text{FREE}_{u,v}, Proj(\tilde{s}))$ 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 $\text{SEND}_{u,v}(p_{data})$ event is enabled in $Proj(s)$ for any edge (u,v). Then $s.|queue_u[v]| > 0$. Thus by Lemma 5.6.6, a $\text{SEND}_{u,v}(p_{data})$ event occurs within t_p time after s. Suppose a $\text{RECEIVE}_{u,v}(p_{data})$ 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 $\text{RECEIVE}_{u,v}(p_{data})$ will occur in t_l time, and $t_l \leq t_p$.

Consider a $\text{FREE}_{u,v}$ action for any edge (u,v). Within t_l time units after s, either a $\text{FREE}_{u,v}$ action will occur or there is a state s' such that $s'.Q_{u,v} \neq nil$. Consider the second case. From Claim B.3.1, 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 $\text{FREE}_{u,v}$ action occurs within time t_l after s''.

Thus a $\text{FREE}_{u,v}$ action will occur within $3t_l$ time units after state s. In particular, this means that a $\text{FREE}_{u,v}$ action will occur within t_p time units of any state s such that $\text{FREE}_{u,v}$ 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. ∎

5.6.6 Tying up the Proof

We now return to the proof of the Local Correction Theorem, Theorem 5.4.3. It follows from Lemma 5.6.23 and Lemma 5.6.22 and Theorem 3.4.2.

5.7 Implementing Local Checking in Real Networks

In this chapter, we described a stabilizing snapshot and reset protocol for link subsystems. This protocol was used to transform an automaton $\mathcal{N}$ that was locally correctable to some predicate L into a UIOA $\mathcal{N}^+$ that stabilizes to the behaviors of $\mathcal{N}(c)|L$ for some constant c. To make the snapshot/reset protocol stabilizing we numbered 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. This solution was advocated by us in [APV91a] and was implemented in a trial implementation on the Autonet [MAM+90].

To keep our model simple, however, we have assumed that all lower bounds (on the time between events) are zero. Thus we have no way to model timers, which are needed to control the interval between phases. While we will not describe the timer based scheme formally, the reader interested in practical applications should be aware of the simplicity of a timer based scheme. Notice also that the maximum value of the timer need only be some small multiple of the worst-case round trip delay on *a single link.* We call such timers *local timers.*

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.

5.8 Summary

The two main contributions of this chapter are the definition of Unit Storage Data Links in Section 5.1.2, and the notion of local checking and correction (Section 5.3 and Local Correction theorem, Theorem 5.4.3).

In a stabilizing setting it is necessary to define Data Links that have bounded storage. First, such models correspond to physical reality. Second, they avoid the theoretical problems with unbounded storage links. We have chosen unit storage links (UDLs) because they are practical (see Section 5.2) and they can be modelled elegantly. We have also defined a stabilizing interface to a UDL. This is done by having the link periodically deliver a free signal (to avoid deadlock) and by having the sender keep a variable that indicates whether the link is free. We hope the UDL model will be used by others.

Intuitively, a protocol is locally checkable if whenever the protocol is in a bad state, some pair of neighbors can detect this fact. Intuitively, a protocol is locally correctable if the protocol can be corrected to a good global state by independently correcting the states of each link subsystem. A link subsystem is just a pair of neighboring nodes and the links between them.

Local checkability is not a very new or surprising idea. For example, many authors have proposed local methods for detecting termination and deadlocks. In a stabilizing setting, the intuitive notion of local checkability was first referred to in a paper by [AKY90]. However, their reference to this concept (during the description of a spanning tree protocol) was brief and intuitive.

Our contribution has been to make precise the notion of local checkability, and to show that this is a useful and pervasive concept. Our definition (Definition 5.3.6) has some subtle aspects: for example, the requirement that each local predicate be closed may not be immediately obvious. Also, we have implemented local checking by doing a snapshot of each link subsystem. Now, a number of practical, self-stabilizing protocols (e.g., [Per85]) do what essentially amounts to local checking in the following way. Periodically each node sends its state to all its neighbors. However, (as we will show in Chapter 8) such periodic sending of state is not always sufficient to do local checking. Periodic sending is sufficient if all local predicates can be separated into what we call (see Chapter 8) one-way predicates. In the general case, a snapshot is required.

The idea of local correction is more unusual. It is perhaps surprising that there are non-trivial protocols with this property. As we will see in Chapter 6, an easy way to ensure local correction is to first build a spanning tree of the network and then do local correction using the tree. Luckily, there is another class of protocols that are locally correctable: these are protocols that work in dynamic networks in which links can fail and recover. We will see an important example of such protocols in Chapter 7.

Local checking and correction is practical. In most real networks, each node sends "keep-alive" packets periodically on every link. The packets used for local checking and correction can be "piggy-backed" on these keep-alive packets.

Chapter 6

Stabilizing Mutual Exclusion and Tree Correction

The main result of Chapter 5 was the Local Correction theorem, Theorem 5.4.3. In this chapter we will consider a simple application of the Local Correction theorem to the problem of *mutual exclusion*. Section 6.1 gives an overview of our stabilizing mutual exclusion protocol. Section 6.2 formally defines our mutual exclusion protocol as a network automaton. Section 6.3 describes how local checking and correction is added to the automaton to create a stabilizing solution to the mutual exclusion solution.

The last two sections of this chapter extract some general principles from the example of stabilizing mutual exclusion. In Section 6.4, we discuss a weaker notion of local checkability called *weak local checkability*. We show that in certain cases, a simple heuristic of removing *unexpected packet transitions* can be used to transform a protocol that is weakly locally checkable into a locally checkable protocol. This heuristic has proved to be quite useful, and is used in later chapters.

Finally, in Section 6.5, we show (informally) an important result, the Tree Correction theorem. This theorem states that any locally checkable protocol on a tree topology can be efficiently stabilized. In other words, if the underlying topology is a tree we can dispense with the need for the (stronger) local correctability condition. This theorem is the counterpart to a similar theorem proved in Chapter 4 for shared memory systems.

6.1 Overview of Token Passing Protocol

Token passing is one way of ensuring mutual exclusion in a network. If there is only one token in a network, a node can go into the critical region when it receives a token. In the I/O automaton model this is typically modelled by adding an output action to every node by which the node can give permission to some external user to go into the critical section, and adding an input action that tells the node when the user has finished with the critical section. For simplicity we ignore this extra piece of modelling which is important for providing a modular interface to other subsystems.

Token protocols that stabilize in time proportional to the height of the tree have been described before by [DolevIM90] in a shared memory setting. However, we believe our protocol is simpler and more transparent. While the mutual exclusion protocol is very simple, its simplicity makes it a good candidate to understand how the general method of local correction developed in Chapter 5 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 modelling the interface to the network automaton that computes the tree, we will (for simplicity) assume that the network graph is a tree. Thus we are dealing with a network automaton of the form $\mathcal{N} = Net(G, N)$ where G is a tree graph. Formally:

Definition 6.1.1 *A tree graph T is a topology graph such that:*

- *T is a rooted tree. (i.e, there is a distinguished node called the root and there is a unique path between any node and the root.)*
- *For every edge (u, v) in G, the leader function $l(u, v) = u$ if u is the parent of v in the tree, and $l(u, v) = v$ if v is the parent of u in the tree.*

Definition 6.1.2 *A tree automaton is a network automaton whose topology graph is a tree graph.*

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

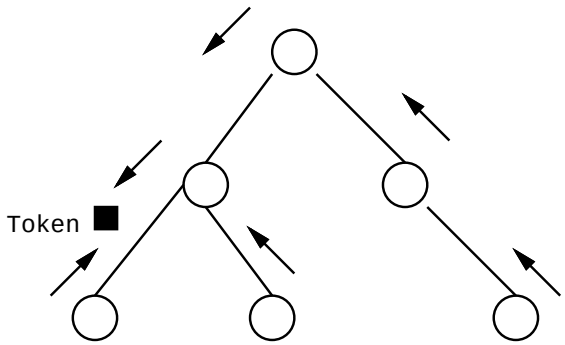

Figure 6.1: Adding a pointer to each node makes token passing on a tree locally checkable.

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 6.1.

6.2 Specification of Token Passing Protocol

Our stabilizing token passing protocol is based on this idea and is specified by $\mathcal{N} = Net(T, N)$ where:

- T is a tree graph.
- Each $u \in V$ is a UIOA of the form described in in Figure 6.2.

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:

- $pointer_u \in neighborSet(u) \cup nil$ (*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 in Chapter 5. To make the token passing automaton a node automaton, we also have to pass the token by first enqueueing the token in the outbound queue for a link. Then, a second separate action is required to send the token from the queue to the link. However, by making N_u a node automaton, we can apply the Local Correction Theorem of Chapter 5.

Notice the code we have given does not specify start states because each N_u is a UIOA .

Lemma 6.2.1 *$N = \{N_u, u \in T\}$ is a set of node automata for T with $P_{data} = \{token\}$.*

. **Proof:** Simple checking of the definitions given earlier. ∎

6.3 Adding Local Checking and Correction

To add local checking and correction, we need to define a link predicate set $\mathcal{L}$ for $\mathcal{N} = Net(T, N)$. 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.

ENQUEUE$_{u,v}$ (*output action to enqueue token in queue for neighbor v*)
  Preconditions:
    $pointer_u = nil$; (* u has token?*)
    $v = Succ_u(last_u)$; (* v is the next neighbor after $last$?*)
  Effect:
    $pointer_u := v$ (* point towards where token is sent*)
    Add $token$ to $queue_u[v]$ (*store token in outbound queue*)

SEND$_{u,v}$($token$) (*output action to send token to neighbor v*)
  Preconditions:
    $free_u[v] = true$ (* link to v free?*)
    $token$ is head of $queue_u[v]$
  Effect:
    $free_u[v] = false$ (* set to false until link says its free*)
    Remove $token$ from head of $queue_u[v]$ (*empty outbound queue*)

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

RECEIVE$_{v,u}$($token$) (*input action, token received from neighbor v*)
  Effect:
    If $pointer_u = v$ then (* token received on expected link?*)
      $pointer_u := nil$ (* accept token*)
      $last_u := v$ (* update $last$*)

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

Figure 6.2: Actions for a node u with respect to a neighbor v in the stabilizing token passing protocol.

Definition 6.3.1 Local Predicates for Mutual Exclusion: *We define the local predicate $L_{u,v}$ of $\mathcal{N}$ 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 6.3.2 *The network automaton $\mathcal{N} = Net(T,N)$ 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 $L_{u,v}$.

```
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 6.3: Code for $f(s|u,v)$, the reset function applied at node u with respect to neighbor 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 $L_{u,v}$. ∎

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 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 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 6.3.

Lemma 6.3.3 *The function f defined in Fig. 6.3 is a local reset function for network automaton $\mathcal{N} = Net(T, N)$ with respect to link predicate set $\mathcal{L}$ and partial order $<$.*

Proof: Consider any edge (u, v) and suppose that u is the parent of v; the reverse case is symmetrical. We check the two conditions in Definition 5.3.7.

- **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 6.3 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 6.3 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 6.3.4 *The network automaton $\mathcal{N} = Net(T, N)$ is locally correctable to L using link predicate set $\mathcal{L}$ and the reset function f defined in Fig. 6.3.*

The following theorem follows directly by applying the Local Correction theorem (Theorem 5.4.3).

Theorem 6.3.5 *Let $\mathcal{N}$ be the set of node automata defined in Figure 6.2, $\mathcal{L}$ be the local predicate set defined in Definition 6.3.1, and f be the local reset function defined in Figure 6.3. Let $\mathcal{N}^+ = Augment(\mathcal{N}, \mathcal{L}, f)$, Then $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(t_p)|L$ in time $t_q \cdot height(T)$, where t_q and t_p are constants.*

Let the symbol $*$ denote any node in the tree. We define the "correct" set of behaviors of a token passing system using a set B. Informally, B is the set of behaviors consisting of phases in which a token is received at a node, then enqueued at the same node, and then passed to a neighbor of the node. Also, we require that a token will be received periodically at every node.

Definition 6.3.6 *We define the set B of correct behaviors of a token passing system as follows. B is the set containing any behavior β that only has actions of $\mathcal{N}$ and such that:*

- *For any u, after any* $\text{RECEIVE}_{*,u}(token)$ *event in β the next event other than a* $\text{FREE}_{*,*}$ *event is an* $\text{ENQUEUE}_{u,*}$ *event.*
- *For any u,v, after any* $\text{ENQUEUE}_{u,v}(token)$ *event in β the next event other than a* $\text{FREE}_{*,*}$ *event is a* $\text{SEND}_{u,v}(token)$ *event.*
- *For any u,v, after any* $\text{SEND}_{u,v}(token)$ *event in β the next event other than a* $\text{FREE}_{*,*}$ *event is a* $\text{RECEIVE}_{u,v}(token)$ *event.*
- *For any u, and any suffix γ of β, a* $\text{RECEIVE}_{*,u}(token)$ *will occur in $c \cdot n$ time after the start of γ, where c is some constant and n is the number of nodes in T.*

We first argue informally that the behaviors of $\mathcal{N}(t_p)|L$ are in B.

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 see 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 of $\mathcal{N}(t_p)|L$, it is quite easy to prove that the behavior corresponding to any execution α of $\mathcal{N}(t_p)|L$ is in B.

For example consider any execution α and any state s_i in α immediately following a $\text{RECEIVE}_{*,u}(token)$ event for some node u. Then it is easy to see that in s_i, $pointer_u = nil$. This predicate will continue to hold until a an $\text{ENQUEUE}_{u,*}$ event occurs. But if $pointer_u = nil$ in a state s, then (since there can be only one token in state s) the only actions enabled are $\text{ENQUEUE}_{u,*}$ or a $\text{FREE}_{*,*}$ event. Similar arguments can be used to show the behavior of α satisfies the next two properties that characterize a behavior in B However, we also need to show the fourth property of a behavior: that for any u, and any suffix γ of α, a $\text{RECEIVE}_{*,u}(token)$ will occur in $c \cdot n$ time after the start of γ, where c is some constant. This can be shown by an inductive argument using the properties of the *Succ* function.

Thus we can show:

Theorem 6.3.7 *Let $\mathcal{N}^+$ be the automaton defined in Theorem 6.3.5. Then $\mathcal{N}^+$ stabilizes to the behaviors in problem B in time $t_q \cdot height(T)$, where t_q and t_p are constants.*

6.4 Removing Unexpected Packet Transitions

Consider the following modification of the token passing protocol described in Figure 6.2. The modification is shown in Figure 6.4. The only difference is that the routine to receive a token at a node u from node v has been changed. The only change is that we no longer check whether $pointer_u = v$ before accepting the token. Let us call the resulting tree automaton $\mathcal{N}^*$.

Assume, however, that we continue to use the definitions of $\mathcal{L}$, L, and $L_{u,v}$ from Definition 6.3.1. Then it is quite easy to prove that $\mathcal{N}^*$ *is* not *locally checkable with*

```
MODIFIEDRECEIVE_{v,u}(token)                    (*token is received from neighbor v*)
  Effect:
    pointer_u := nil                                          (* accept token*)
    last_u := v                                               (* update last*)

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

Figure 6.4: Modified Code for a node u in a token passing protocol. The remaining code is identical to the code in Figure 6.2

respect to L and $\mathcal{L}$. This follows from the fact that $L_{u,v}$ is not a closed predicate of $\mathcal{N}^*$.

Despite this it is not hard to prove that the behaviors of $\mathcal{N}^*|L$ are exactly the behaviors of $\mathcal{N}|L$. In fact, if we are allowed the luxury of specifying initial states, $\mathcal{N}^*|L$ is a "natural" IOA to solve the token passing problem. Suppose a protocol designer has started with $\mathcal{N}^*|L$ and now wishes to construct a UIOA that stabilizes to the behaviors specified by the token passing problem. It is interesting to note that this can be done by the following two step process:

- Transform $\mathcal{N}^*$ into $\mathcal{N}$ by adding the extra check shown in Figure 6.2.
- Transform $\mathcal{N}$ into $\mathcal{N}^+$ as shown earlier.

We would like to abstract this process. First, note that the extra check added in going from $\mathcal{N}^*$ into $\mathcal{N}$ essentially amounts to the following heuristic: *if we receive an "unexpected" packet p at node u from node v, then we do not process p.* Notice that in $\mathcal{N}$, a token received from v when $pointer_u \neq v$ (see Figure 6.5) is certainly "unexpected". We will formalize this intuitive notion of an "unexpected" packet below. However, for the present it is important to intuitively understand why such checks for unexpected packets are useful. Consider a transition $(s, \pi, \tilde{s})$ of $\mathcal{N}$ such that s satisfies $L_{u,v}$ but not $L_{u,w}$. Then it is quite possible that there is an "unexpected" packet on channel $Q_{w,u}$ in state s. By adding checks for such packets in $\mathcal{N}$, we can ensure that $\tilde{s}$ satisfies $L_{u,v}$. Also note that these checks do not affect the "correct" executions of $\mathcal{N}|L$.

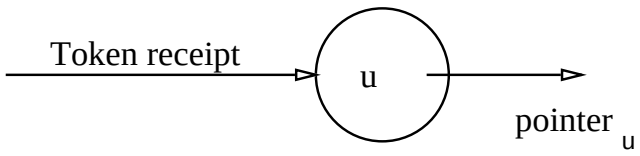

Figure 6.5: Receiving a token on a link that is not being pointed to is an unexpected packet transition. In general, an unexpected packet transition is a packet reception that could never have occurred if the receiving link subsystem was in a good state.

We now formalize these observations. First, we define the notion of weak local checkability. Intuitively, we remove the requirement that that each $L_{u,v}$ be a closed predicate and instead require only that L is a closed predicate.

Definition 6.4.1 *A network automaton $\mathcal{N}$ is weakly checkable for predicate L using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{L}$ is a link predicate set for $\mathcal{N}$ and $L \supseteq Conj(\mathcal{L})$.*
- *For any transition $(s, \pi, \tilde{s})$ of $\mathcal{N}$, if $s \in L$ then $\tilde{s} \in L$*

Ideally, we would like to prove that any weakly checkable protocol can be transformed into an "equivalent" locally checkable protocol. While we do not know how to do this in general, we can can obtain such a result if the automaton is also *locally extensible*. Intuitively, an automaton is locally extensible with respect to a link predicate set if any pair of "good" adjacent link subsystem states can be extended to form a "good" state of the entire automaton.

Definition 6.4.2 *A network automaton $\mathcal{N} = Net(G, N)$ is locally extensible with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ if the following condition is true:*

For any two adjacent edges (u, v) and (v, w) in G, if $x \in L_{u,v}$ and $y \in L_{v,w}$ then there is some state $s \in Conj(\mathcal{L})$ such that $x = (s|u, s|(u, v), s|(v, u), s|v)$ and $y = (s|v, s|(v, w), s|(w, v), s|w)$.

To transform a locally extensible and weakly checkable protocol $\mathcal{N}^*$ into an "equivalent" locally checkable protocol, we will add checks to $\mathcal{N}^*$ for "unexpected" packets. We formalize this notion of an "unexpected packet" (see Figure 6.5) as follows:

Definition 6.4.3 *Consider a network automaton $\mathcal{N}$ with link predicate set $\mathcal{L} = \{L_{u,v}\}$ and some pair of neighbors u, v in $\mathcal{N}$. We say that a transition $(s, \pi, \tilde{s})$ is an unexpected packet transition at u with respect to v if:*

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

For example, in Figure 6.2, a $(s, \text{RECEIVE}_{v,u}(token), \tilde{s})$ transition with $s.pointer_u \neq v$ is an unexpected packet transition at u with respect to v. We can now state a simple theorem.

Theorem 6.4.4 *Consider a network automaton $\mathcal{N}^* = Net(G, N)$ that is weakly checkable for predicate L with respect to predicate set $\mathcal{L}$. Suppose further that $\mathcal{N}$ is locally extensible with respect to $\mathcal{L}$. Then it is possible to construct another automaton $\mathcal{N}$ such that:*

- *$\mathcal{N}$ is an automaton for graph G and the executions of $\mathcal{N}|L$ are identical to the executions of $\mathcal{N}^*|L$*
- *$\mathcal{N}$ is locally checkable for predicate L with respect to predicate set $\mathcal{L}$.*

Proof: We transform $\mathcal{N}^*$ into $\mathcal{N}$ by replacing all unexpected packet transitions $(s, \pi, \tilde{s})$ in $\mathcal{N}^*$ by the null transition (s, π, s). Then we use the local extensibility property to show each $L_{u,v}$ is a closed local predicate of $\mathcal{N}$. ∎

6.5 Tree Correction Theorem

The reader who has read Chapter 4 might suspect that any locally checkable protocol on trees can be made locally correctable. Thus for tree automata it seems plausible that we do not need the stronger hypothesis that the tree automaton be locally correctable. This is indeed true. Compare the following theorem to Theorem 4.3.1 (but be aware that Theorem 4.3.1 applies to shared memory tree automata) and the Local Correction theorem (Theorem 5.4.3).

Theorem 6.5.1 Tree Correction: *Consider any tree automaton $\mathcal{T} = Net(T, N)$ that is* locally checkable *for L using link predicate set $\mathcal{L}$. 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 $\tilde{c} \cdot height(T)$.*

The proof of this theorem is extremely similar to that of Theorem 4.3.1 of Chapter 4. However, since we can no longer do snapshots and resets atomically, we need to use the local snapshot and reset protocols defined in Chapter 5. Thus we have to add actions for local checking and correction as in the proof of the Local Correction Theorem, Theorem 5.4.3. However, there are two differences. We will assume that for every (u, v) subsystem in which v is the parent of u, the child u performs the checking/correction. The local snapshot protocol is identical to the protocol of Chapter 5, but the local reset protocol is a little different. This is sketched in Figure 6.6. The figure should be compared with the right hand diagram in Figure 5.9.

The basic idea is that the reset response carries the state b of the parent at the instant the response was sent. When the child gets the response, the child sets its state to $f(b)$, where f is a function that we describe next. Basically, f is chosen such that for every state b of the parent node v, $(f(b), nil, nil, b) \in L_{u,v}$. In other words, f is chosen so that we can reset the link subsystem to a good state by only changing the state of the child node. Of course, that is the basic idea in the proof of Theorem 4.3.1. The only tricky part is to argue that we can find such a function f. The proof is again similar to the proof of Theorem 4.3.1: we first normalize the original protocol $\mathcal{T}$ to get rid of "useless" node states that can never occur in global states in which all local predicates are true. We then show that the required function f exists for the normalized protocol.

The Tree Correction theorem is not a corollary of the Local Correction theorem. Recall in Chapter 5 that when we applied a local reset function to the state of leader node u with respect to node v, the resulting state of node u can only depend on the previous state of node u. However, in the proof of the Tree Correction theorem we require the the resulting state of node u to depend on the previous state of node v!

Finally, we note that we could have derived the stabilizing mutual exclusion protocol by showing that it was locally checkable and then using the Tree Correction theorem directly.

6.6 Summary

In Chapter 4, we showed (in a shared memory model) that any locally checkable protocol on a tree could be stabilized in time propostional to the height of the tree.

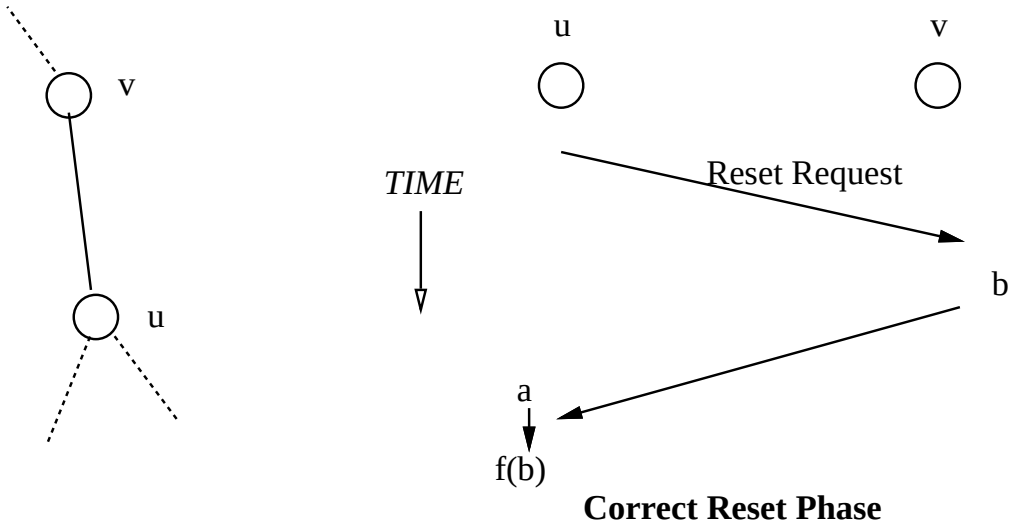

Figure 6.6: Sketch of a reset phase used for Tree Correction. Node v is the parent of node u in the tree.

This chapter shows that this theorem can be extended to the message passing model (Tree Correction theorem, Theorem 6.5.1). We also described a simple application – the problem of token passing on a tree. A stabilizing solution to this problem can be derived using either the Local Correction or Tree Correction theorems.

The token passing problem suggests a simple strategy for stabilizing protocols. First, we try to add a small amount of state to make the protocol locally checkable. Recall that we added a pointer to each node for this purpose in the token passing protocol. Then we combine the original protocol with another protocol that computes a spanning tree. Finally, we do local correction on the resulting spanning tree. Although we have not done so, it is important to formally describe how an arbitrary protocol P could be composed with a spanning tree protocol so that the net effect is the same as if P were running on the final tree.

Local checkability requires that each local predicate be a closed predicate. Removing unexpected packet transitions (Figure 6.5) is a useful heuristic that is often sufficient to ensure that each local predicate is closed.

The token passing protocol on a tree can be generalized to passing a constant number of tokens on a tree. In this case, we replace the pointer variable $pointer_u$ at each node u by a variable $token_count_u[v]$ (one for each neighbor v) that keeps track of the number of tokens that are in the direction of neighbor v. The local predicates

have also to be suitably modified.

Chapter 7

Stabilizing Network Reset

The stabilizing network reset protocol described in this chapter is the link between the previous two chapters (which were about Local Correction) and the last two chapters of the thesis (which are about Global Correction).

The major service provided by a network reset protocol is *synchronization* of the nodes in a network. In the first section of this chapter, we informally introduce the concept of synchronization, and discuss why this service is useful. In Section 7.2 we review existing reset protocols. Then in Section 7.3 we specify the reset problem. Previous specifications of reset protocols have been state-based. Our specification of the reset problem is novel in that it is based on external behaviors. In the next section (Section 7.4), we give an overview of our reset protocol. Then in Section 7.5, we formally specify our reset protocol using a reset automaton.

Sections 7.6 and 7.7 are devoted to showing that the reset automaton is a *stabilizing* solution to the reset problem. We do this using the Local Correction theorem developed in Chapter 5. We show that the reset automaton is locally checkable by exhibiting a set of closed local predicates for the reset automaton. Then we show that the reset automaton is locally correctable by showing that the local predicates are independent – i.e., the partial order that formalizes the dependency relation between the predicates is the trivial partial order. Thus the reset protocol stabilizes in constant time. Next, (in Section 7.7), we show that once the reset automaton is in a state that satisfies all local predicates, then the behaviors exhibited by the reset automaton are indeed the behaviors specified by the reset problem. This completes the proof that the reset automaton is a stabilizing solution to the reset problem.

The last two sections of the chapter try to abstract some general principles based on the example of the stabilizing reset protocol. The reset protocol in this chapter is based on an existing non-stabilizing reset protocol [AAG87] that works in networks where links can fail and recover. Section 7.8 suggests that this is no accident – locally checkable protocols that work in networks where links can fail and recover are likely to also be locally correctable.

Because this chapter is very long, we offer two suggestions for reading. First, constantly consult the roadmaps at the beginning of the chapter and each long section. Second, it is hard to appreciate the specification of the reset problem until one sees why it is useful. Chapters 8 and 9 describe important applications of the reset protocol. Readers may prefer to read Chapter 8 after reading the specification in Section 7.3 and before reading the rest of Chapter 7.

7.1 Synchronization

A reset protocol is used to *synchronize* all the nodes in a network. Before we describe what we mean by synchronizing *all* the nodes in a network, it is helpful to understand a form of synchronization between *a pair* of nodes in a network. Such synchronization is provided by a Data Link protocol [Spi88a]. We will then see that in essence a network reset protocol generalizes the guarantees of a Data Link protocol to *multiple nodes* in a network.

7.1.1 Data Link Synchronization

Consider a pair of neighboring nodes in a network, say u and v. Suppose the physical link between two nodes in a network can periodically crash and recover. The Data Link protocol is responsible for providing notification (as to whether the link is up or down) to the users at u and v. Thus at each node, the Data link protocol issues Link up and Link down actions to signify that the link is operational or not operational respectively. If the network is asynchronous, it is impossible to provide a Link down event (or Link up event) at exactly the same instant at both u and v. But if Link up and down events are reported independently (and possibly at different times) at both ends, the Data Link must provide some additional functions to *synchronize* u and v.

The synchronization requirements for a Data Link protocol can be stated elegantly [Spi88a] as follows. First, for node u (or node v) we can define an *operating interval* at node u (or at node v) to be the time from a Link up event at that node until the next Link down event at that node. If an operating interval does not end with a Link down event we say that the interval is a *final interval.* Thus each execution of the Data Link protocol induces a set of operating intervals at both nodes. Then for synchronization, we require that there is a symmetric relation between intervals at the two nodes (called a *mating* relation in [AE86]) such that:

- An operating interval can be mated to at most one other operating interval.
- Suppose an operating interval α at u is mated to an operating interval β at v. Then the sequence of messages received by v in β must be a prefix of the sequence of messages sent by u in α. (This is often called the prefix property of Data Link protocols.) Also, if α is a final interval, then so is β and in this case the sequence of messages received by v in β must be identical to the sequence of messages sent by u in α.
- Suppose an operating interval α at u is mated to an operating interval β at v and an operating interval α' at u is mated to an operating interval β' at v. Then if α' occurs later than α then β' occurs later than β.

The mating relation for two nodes u and v is sketched using the time-space diagram shown in Figure 7.1. We have depicted Link up events by horizontal lines. The Link down events between Link down events are depicted by crosses. An arrow from v to u depicts a packet sent by v that is successfully delivered at u, and vice versa. An arrow from v that does not reach u is a packet sent by v that is *not* delivered at u. In the figure, the second operating interval at u mates to the second operating interval at v. Also, the third operating interval at u mates to the fourth operating interval at v and the two intervals are final intervals. Notice that the sequence of packets received by u in its second interval is a prefix of the sequence of packets sent by v in its second interval. Also notice that all packets sent in the two final intervals are delivered.

Why does this mating relation provide a useful form of synchronization? It does so because the synchronization relation guarantees that the behavior of a node during an operating interval is *what might have occurred in some asynchronous execution of the*

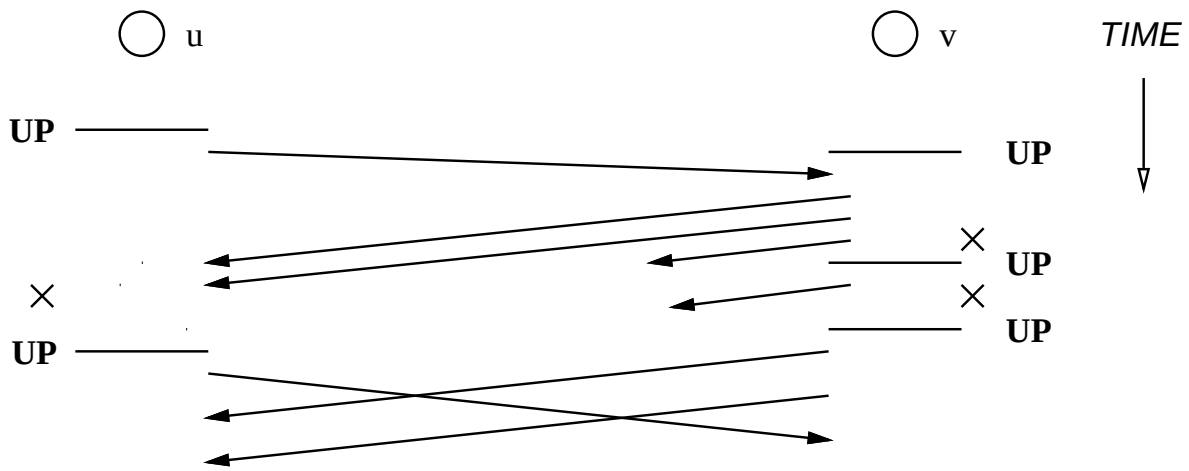

Figure 7.1: Illustrating the Mating Relation

Data Link protocol in which there were no link failures. This is a crucial abstraction that allows users of the Data Link protocol to deal with failures in a simple way.

Almost identical forms of synchronization are provided by virtual circuit protocols ([Spi88a]) and transport protocols. Thus all these protocols essentially synchronize two nodes in a network.

7.1.2 Network Synchronization

The synchronization provided by a network reset protocol is a generalization of the synchronization guarantees of a Data link. A reset protocol synchronizes *all* the nodes in the network. Informally, the reset problem [Fin79] is to design a reset service that can be superimposed on any other distributed protocol P. The reset may be invoked at any node, and its effect is to output a signal at all the nodes of the system in a consistent way. We use Σ-messages to denote the messages sent and received by protocol P.

By consistent, we mean the following. As in the Data Link protocol, the reset protocol induces signal intervals at each node (i.e., intervals between consecutive signal events at a node). Then for each pair of neighbors u and v we require that the signal intervals at u and v can be mated as described earlier. For example, if we replaced the Link up events with Signal events in Figure 7.1 and ignored Link down events, then

Figure 7.1 could equally well describe the pairwise mating relation provided by a reset protocol. Notice, however, that in a Data Link the Link up events occur independently on each link and so the the operating intervals on each link adjacent to a node u can be different. However, in a Reset protocol, the Signal events induce signal intervals on a per node basis.

For example this means that the sequence of Σ-messages *sent* by any node u to any neighbor v after the last signal at u must be equal to the sequence of Σ-messages *received* by v after its last signal.

Why is this called a reset service? Suppose S_u defines the set of "legal" start states for every node u. Suppose the "legal executions" of P are those executions in which the initial state of every node u belongs to S_u and the initial state of every channel is a state in which the channel contains no Σ-messages. It's natural to define the legal states of P as those states that occur in legal executions of P. Suppose we now superimpose a reset service over P. Suppose also that whenever a node u receives a signal, we locally reset the state of node u (i.e., the local state of protocol P at node u) to some state in S_u. Then after every node has received its last signal, protocol P is in a legal state. In other words, the signals provide a consistent time point for every node u such that we can *globally reset* protocol P to a legal state by *locally resetting* each node u at its time point. The time point for each node is the instant it receives a signal.

Thus a global reset service is much like the reset button on a computer. After the reset button is pushed, the computer is restored to a "good" state. However, globally resetting a network to a "good" state is much more challenging than resetting a single node. In a network, reset requests can arrive at any node and the signals must be delivered at every node in a consistent fashion. Ideally, we would like the signals to be delivered to every node at the same instant. However, since this seems to be impossible in a distributed system, we settle for delivering the signal in a consistent manner (see above).

Why is a reset service useful? It was introduced in [Gal76, Fin79] as a tool for converting any protocol that works in a so-called *static network* to work in a so-called *dynamic network*. A static network, as the name suggests, is a network in which the topology of the network remains fixed during the execution of the algorithm. A dynamic network, by contrast, is a network in which nodes and links can crash, thereby changing the topology. However, it is assumed that topology changes eventually stop

and that some node in the final topology can detect the fact that there has been a topology change. Roughly, the idea behind [Fin79] is that any node that detects a topology change makes a reset request. If successful, the reset request results in restarting the static protocol. This methodology is quite practical. For instance, the Autonet local area network uses a version of [Fin79] to cope with failures.

Besides its use in dynamic networks, a reset protocol is also useful in a stabilizing setting. As we show in Chapters 8 and 9, *a stabilizing reset protocol is an important tool for the design of other stabilizing protocols.* Notice that in order to use the reset service in dynamic networks [Fin79], some node must detect the last topological change. More generally, suppose that any bad state of a network protocol can be detected locally by some node. This corresponds to what we have called a locally checkable protocol in Chapter 5. As we will see in Chapter 8, any locally checkable protocol can be stabilized using a stabilizing reset protocol. Intuitively, our idea is similar to that of [Fin79]. We add actions to each node to make a reset request if the node locally detects a bad state of the network.

The technique of using a reset protocol to stabilize a locally checkable protocol is quite different from the techniques developed in Chapter 5 and 6. In Chapter 5, a locally checkable and correctable protocol is stabilized by doing *independent local resets* of each link subsystem. In Chapter 6, a locally checkable protocol on a tree is also stabilized by doing *independent local resets* of each link subsystem.

By contrast, we can stabilize a locally checkable (but perhaps not locally correctable) protocol by doing a *coordinated global reset* of the entire network. As one might guess, there is a performance penalty in using a network reset. The stabilization time of a solution that uses local correction is proportional to the height of the underlying partial order (see Local Correction Theorem, Theorem 5.4.3). The stabilization time of a solution that uses tree correction is proportional to the height of the tree (see Tree Correction Theorem, Theorem 6.5.1).

However, the stabilization time of a solution that uses global correction is proportional to the number of nodes in the network. Because the height of the partial order (or the height of a tree) typically is smaller than the number of nodes, we have the following rule of thumb. If a protocol is locally correctable or runs on a tree, then it pays to use the techniques of Chapter 5 or 6. However, if a locally checkable protocol cannot be shown to be locally correctable, then the network reset approach provides a (perhaps less efficient) stabilizing solution. It is perhaps elegant that the network

reset protocol is itself stabilized using the local reset approach of Chapter 5.

In this chapter we introduce the most efficient known stabilizing network reset protocol. We do so by applying the method of local checking and correction of Chapter 5 to an existing reset protocol described in [AAG87] The space overhead of the protocol is logarithmic. Our reset protocol stabilizes in constant time.

7.2 Existing Solutions

In Chapter 4, we described a stabilizing reset protocol due to Arora and Gouda [AG90]. Their protocol was described in terms of a shared memory model but it appears that it can be adapted to work in our message passing model. [AG90] also describes a stabilizing protocol to build a spanning tree of the network. For the spanning tree protocol, it is assumed that processes have unique identifiers, and that there is some *a priori* bound K on the number of nodes in the network. The IDs and K cannot be corrupted by transient errors. The stabilization time of the spanning tree protocol is $O(K)$, where K is a non-volatile bound on the number of nodes in the network. Their protocol will also work correctly if K is an upper bound on the diameter of the final network. However, in a network in which the topology can change arbitrarily, often the only reasonable bound on the diameter of the network is a bound on the number of nodes in the network. Secondly, as we will see in Chapter 8, their spanning tree protocol is based on a routing algorithm that works poorly in practice.

Katz and Perry [KP90] describe a stabilizing reset protocol. Their approach requires the election of a leader and the stabilization time of their approach is $O(n^2)$ where n is the number of nodes in the network.

On the other hand, the reset protocol we describe does not require node IDs, and takes $O(n)$ time to stabilize, where n is the *actual* number of nodes in the network. Our protocol does not require the computation of a spanning tree. Thus it can be used to build a stabilizing spanning tree protocol as we show in the Chapter 8.

7.3 Specifying the Desired Behaviors of a Reset Protocol

This section is divided into four subsections. First, we describe the external interface to the reset protocol. Then, we give an overview of the specification and some of its difficulties. After this motivation, we formally specify the reset problem and then briefly discuss alternative specifications.

7.3.1 Interface to Reset Protocol

We first describe the external interface to the reset protocol.

A reset service is modelled as a network automaton $\mathcal{R} = Net(G, N)$. The external interface for any node u in the network is shown in Figure 7.2. We have the usual interfaces to send and receive *packets* between neighbors as described in Chapter 5. However, in addition each node also has interfaces to send and receive *messages* on behalf of external users of the reset service. We assume that every message m is drawn from some message alphabet Σ, and that $\Sigma \subset P_{data}$, where P_{data} is the data packet alphabet used by the network automaton. Intuitively, the messages sent by users to the reset service will be relayed between nodes of the network using packets.

Thus node u also has an input action $\text{SENDM}_{u,v}(m)$ by which an external user can send a message to neighbor v. Similarly node u has an output action $\text{RECEIVEM}_{v,u}(m)$ by which the reset service can deliver a message m from the user at node v to the user at node u. There is also a $\text{FREEM}_{u,v}$ output action that is used by the reset service to indicate that it is ready to accept another message from node u to node v. Thus the external interface between a reset service and its users essentially mimics the interface offered by a UDL (see Chapter 5) except that packets are replaced by messages. This is an important property that will be exploited later.

However, the interface between a user and a reset service is richer than the interface between a user and a set of UDLs. This is because the reset service at node u offers two additional actions: an input action REQUEST_u and an output action SIGNAL_u. Intuitively, the REQUEST_u action is used by the user at node u to request a network reset, and the SIGNAL_u action is used to inform the user at node u that a reset has

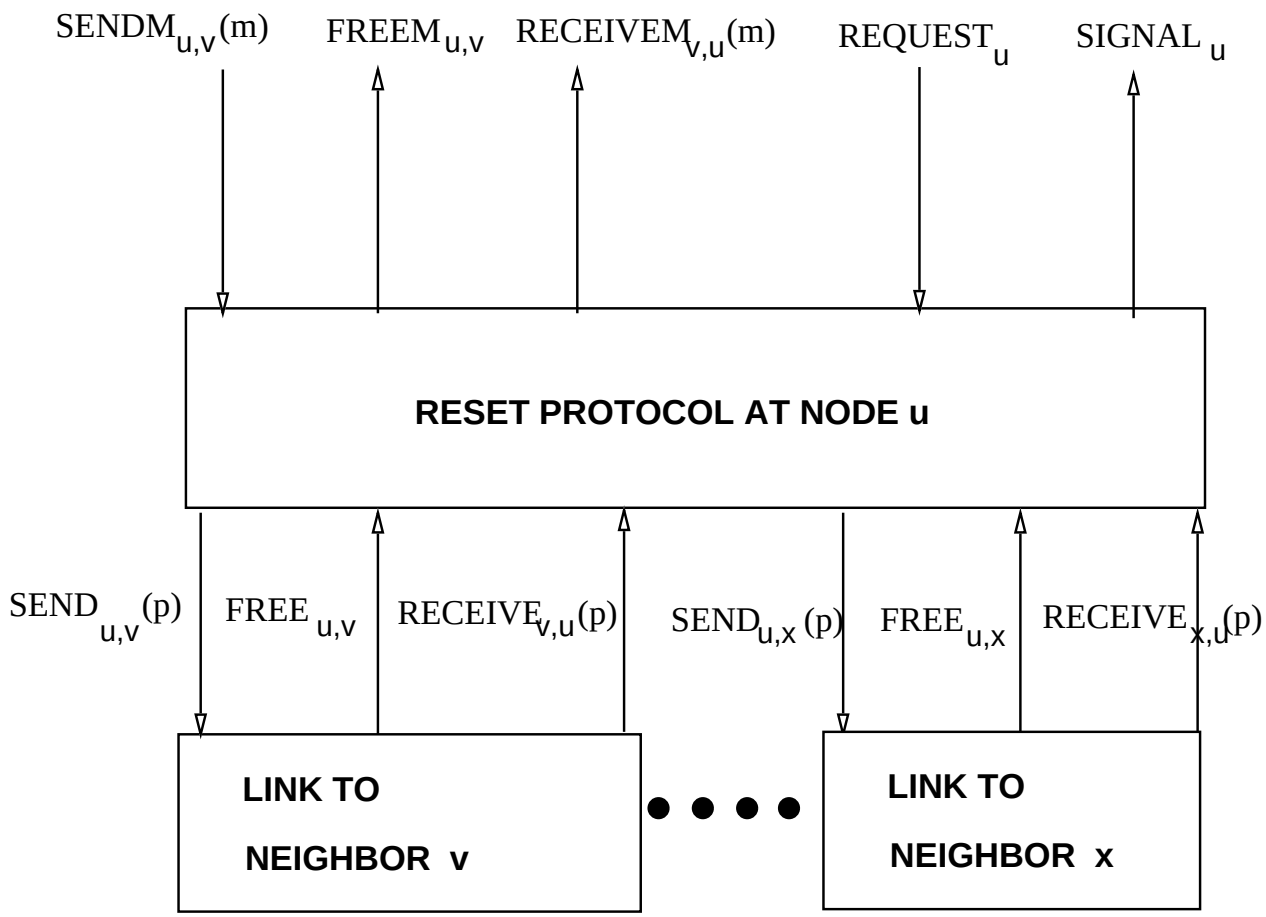

Figure 7.2: Interface specification for reset service

been completed. We will refer to any event of the form SIGNAL_u as a *signal event* and any event of the form REQUEST_u as a *request event.*

7.3.2 Difficulties in Specifying the Reset Problem

The ideal behaviors of a non-stabilizing reset protocol can be specified in terms of three properties: *timeliness, consistency, and causality.* Intuitively, a behavior is timely if, in the absence of reset requests, free events are delivered in constant time and sent messages are delivered in constant time. A behavior is consistent if there is a symmetrical *mating* relation between signal intervals at neighbors. Finally, a behavior is causal if reset signals are only caused by reset requests and reset requests result in reset signals.

As in a UDL, we cannot guarantee that any message sent by a user from say u to v will be received at v unless (see Figure 7.2) this message is sent after u performs a $\text{FREEM}_{u,v}$ action. If this is true (and no other message is sent from u to v between the free action and the send action), then we will say that the send action is *safe*. We will only require delivery of messages corresponding to safe message send events. The specification will allow other messages to be dropped.

Any specification of the behaviors of a stabilizing reset protocol has to take into account three anomalies that do not occur in the specification of a non-stabilizing reset protocol:

- The first message sent on any link may not be *safe* in that it may not be preceded by a free event.
- Some of the initial messages that are delivered may be *abnormal* in that they do not correspond to any messages sent. It appears that no stabilizing reset protocol can guarantee that there is some suffix of every behavior that contains no abnormal message delivery.[1]
- Some of the initial signal actions may not be "caused" by any reset request.

[1] A stabilizing reset protocol can begin in a state in which all links can have arbitrary messages stored. There can be executions that contain no state in which all links are empty of messages. Any suffix of such an execution will have abnormal message deliveries.

We handle these difficulties as follows. We add the following condition to the timeliness property: within linear time of the start of any behavior of the reset protocol, all received messages are *normal* – i.e., correspond to some message sent. We weaken the mating relation so that it is possible to receive abnormal messages in a signal interval – however, normal messages received in a signal interval must have been sent in a mated interval. Finally, we relax the consistency property and ask only that causality holds in linear time after the start of a behavior. Precise definitions are given in the next subsection.

7.3.3 Formal Specification of Reset Problem

Recall the following notation. When we say that an event a_j occurs within time t in β we mean that $a_j.time \leq \beta.start + t$. When we say that a time t is a constant, we mean that $t = ct_l + c't_n$ where c, c' are some scalar constants and t_n and t_l are the default node and link delays respectively. In this and following chapters, we will also use the following notation borrowed from complexity theory. When we say that $t = O(n)$, we mean that $t \leq cn$, where c is some constant time and n is the number of nodes in the graph G. This reflects a linear dependence on the size of the input, if we consider the input to be the network graph. If we say that an event a_i occurs within $O(n)$ time in a behavior β, we mean that $a_i.time - \beta.start = O(n)$. If we say that an event a_j occurs within $O(n)$ time after an earlier event a_i in behavior β, we mean that $a_j.time - a_i.time = O(n)$; in this case, we will also say that event a_i occurs within $O(n)$ time *before* event a_j in behavior β. Sometimes we will say linear time to mean an interval of time that is $O(n)$ in duration.

As in a UDL, we cannot guarantee that any message sent by a user from say u to v will be received at v unless (see Figure 7.2) this message is sent after u performs a $\text{FREEM}_{u,v}$ action. If this is true (and no other message is sent from u to v between the free action and the send action), then we will say that the send action is *safe*. We formalize this restriction by defining what it means for a message send event to be *safe*.

Definition 7.3.1 *Consider any behavior β of a reset protocol that has the interface shown in Figure 7.2. We say that an action $a_j = \text{SENDM}_{u,v}(m)$ is safe in β if:*

- *There is some $\text{FREEM}_{u,v}$ action in β before a_j and*

- *No other action of the form* $\text{SENDM}_{u,v}(*)$ *occurs between* a_j *and the* $\text{FREEM}_{u,v}$ *action.*

Clearly we would eventually like every message that is *received* at u from say v to correspond to some message *sent* from v to u. We will require that there is at most one message in transit from v to u. This will make it easy to make a correspondence between received and sent packets. Thus:

Definition 7.3.2 *We say that event* $a_k = \text{RECEIVEM}_{v,u}(m)$ *in* β *is a normal receive at* u *from* v *iff:*

- *There is some* $a_j = \text{SENDM}_{v,u}(m)$ *in* β *that occurs before* a_k *in* β *and*
- *There is no* $\text{RECEIVEM}_{v,u}(*)$ *event between* a_j *and* a_k *in* β.

We will refer to the earliest $\text{SENDM}_{v,u}(m)$ *in* β *that occurs before* a_k *(and satisfies the above properties) as the send corresponding to* a_k.

Notice that if the sender ignores the free notification and sends a message m several times before it is received, we make a correspondence between the receive and the earliest send event.

Specifying Timeliness

Next, we formalize the timeliness property. We we say that a behavior is timely if it satisfies four conditions for every pair of neighbors u and v. First all receive events that occur after linear time in β are normal. In other words, after at most $O(n)$ time, each packet received corresponds to some packet sent. Also any normal receive event occurs within $O(n)$ time after the corresponding send event. This is shown in Figure 7.3.

The second condition is that either the user at u periodically receives a free action indicating that the link to v is free or else a signal event occurs periodically at either u or v. Third, any message sent by v to u is either delivered in constant time or else a signal event occurs (at either u or v) after the message is sent. The essence of the second and third conditions is that, in the absence of signal events, free events are

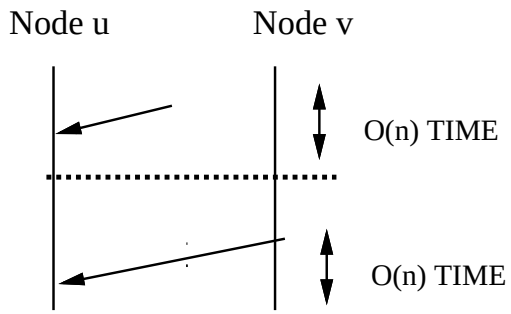

Figure 7.3: Normal message receipt after linear time: There is a send action at v corresponding to any message received at u after $O(n)$ time. Also any normal receive event occurs within $O(n)$ time after it is sent.

delivered periodically and sent messages are delivered in constant time. Intuitively, signal events are caused by reset requests; if reset requests are made continuously, then the reset protocol cannot guarantee periodic free events or the delivery of messages.

The fourth timeliness condition is important (for example in Chapter 8) in applications. It says that signals cannot keep occurring at u without also occurring at a neighbor v. More precisely, if a signal event occurs at u then a signal event must occur at v in linear time before or after the signal event at u. However, because the reset protocol can start in an arbitrary state, we relax this requirement and only ask that this property hold after a linear amount of time.

Definition 7.3.3 *We say that a behavior β is timely if for every pair of neighbors u, v:*

1. **Normal Receipt of Messages**: *There is some constant c such that every every receive event that occurs at time greater than $\beta.start + c \cdot n$ is normal. Also if a_j is any normal receive event and a_i is the send corresponding to a_j, then a_j occurs within $O(n)$ time after a_i.*

2. **Periodic Free Events**: *Consider any t-suffix γ of behavior β. Then in γ either a $\text{FREEM}_{u,v}$ occurs within constant time, or a signal event occurs at u within $O(n)$ time, or a signal event occurs at v within $O(n)$ time.*

3. **Timely Message Delivery**: *Suppose a_j is a safe $\text{SENDM}_{u,v}(m)$ event in β. Then after a_j either a $\text{RECEIVEM}_{u,v}(m)$ occurs within constant time, or a signal*

event occurs at u within $O(n)$ time, or a signal event occurs at v within $O(n)$ time.

4. **Signals at a Node induce Signals at Neighbors**: *There is some constant c such that for every* SIGNAL$_u$ *event a_j that occurs at time greater than $\beta.start + c \cdot n$ there is a* SIGNAL$_v$ *event that occurs in linear time before or after a_j.*

Specifying Consistency

Before we formalize the consistency property, we formalize the notion of a signal interval at a node in a behavior.

Definition 7.3.4 *Consider a behavior β. A signal interval at node u in β is a contiguous subsequence of β that:*

- *Begins with either the start of β or a* SIGNAL$_u$ *event and*
- *Ends with either the first* SIGNAL$_u$ *event that occurs after the start of the interval or (if there is no such* SIGNAL$_u$ *event) the interval ends with the end of β. In the latter case we call the signal interval a final interval.*

Thus any behavior induces signal intervals at each node. We now specify the consistency condition by requiring a mating relation between signal intervals. However, the mating relation is weaker than the relation for Data Links because it does not require the prefix property. The third property is a little tricky. Consider Figure 7.4. Suppose the send at v is safe and is followed by a free action at v that is in the same signal interval at v. Then we require that the sent message is delivered and the delivery event occurs between the send and the free events. In essence, this states that all messages (except possibly the last message) sent safely in a signal interval at v are delivered at u.

Definition 7.3.5 *We say that a behavior β satisfies the consistency property if for every pair of neighbors u, v there is a symmetric relation (called a* mating *relation) between signal intervals at u and v such that:*

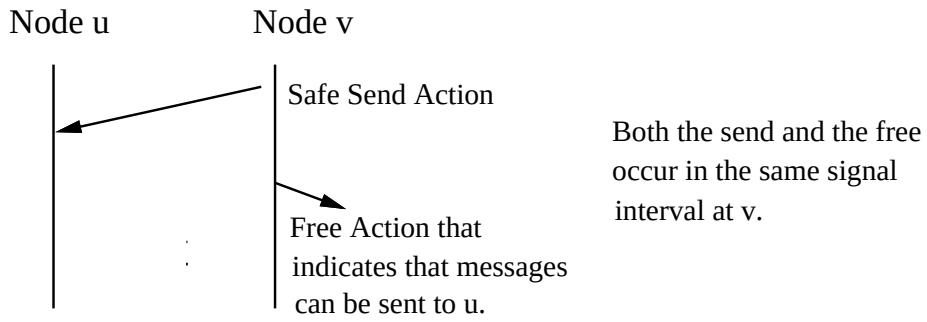

Figure 7.4: Successful sending of messages: between the sending of a message and the free action that indicates that the next message can be sent, either the message is delivered or a signal occurs at the sender.

1. **At most one mate**: *A signal interval at u can be mated to at most one signal interval at v.*

2. **Signal Intervals that communicate are mates**: *Let a_k be any normal receive event at u from v in β and let a_m be the send event corresponding to a_k. Then the signal interval at u containing a_k is mated to the signal interval at v containing a_m.*

3. **Successful Sending of Messages**: *Consider any safe* $\text{SENDM}_{v,u}(m)$ *event a_j and a later* $\text{FREEM}_{v,u}$ *event that occur in a signal interval at v. Then between these two events in β there must be a normal* $\text{RECEIVEM}_{v,u}(m)$ *event a_k such that a_j is the send corresponding to a_k.*

4. **Mating of Final Signal Intervals**: *A final interval at u can only mate to a final interval at v.*

5. **Mating Relation Preserves Temporal Ordering**: *Suppose a signal interval S_u at u is mated to a signal interval S_v at v and a signal interval S'_u at u is mated to a signal interval S'_v at v. Then if S'_u occurs later than S_u then S'_v occurs later than S_v.*

Suppose S_u and S_v are signal intervals at u and v respectively that are mates. Notice that as compared to a Data Link specification, we have weakened the requirements for a mating relation: we no longer require that the sequence of message received by v from u is a prefix of the sequence sent from u to v. *However, if all received messages*

are normal and all sent messages are safe, then the third consistency condition does imply the prefix property. The third and fourth consistency conditions also imply that if S_v is a final interval, and if all received messages are normal and all sent messages are safe, then the two sequences are *identical.*

The reader may wonder whether it is sufficient to specify consistent behavior only in final intervals. In that case, the consistency condition is much simpler. However, if the stabilizing reset protocol is to be used as a tool to build other stabilizing protocols (which is what we do in the next two chapters), then we claim that the reset protocol *must* make some guarantees during non-final intervals. A typical user of the reset protocol will be constantly doing some form of checking (see Chapter 8 for example) and will make a reset request if the checking detects a violation. But if the reset protocols exhibits arbitrary behavior during non-final intervals, then the user may always detect violations. These in turn lead to continuous reset requests which prevent the forming of a final interval. In other words, final intervals are only guaranteed if the user stops making reset requests; but users may only stop making reset requests if the reset protocol guarantees some form of consistency during non-final intervals. As another example, some user protocols may periodically make reset requests to start a new phase of the protocols. Such protocols (Chapter 9 describes an example of such a protocol) never stop making reset requests!

Specifying Causality

A causal behavior satisfies two intuitive conditions: that signal events are only caused by reset requests and reset requests result in signal events. Ideally, any signal event must be preceded by a reset request that occurs a linear amount of time before the signal event. Notice that this guarantees that all signal events will disappear in linear time after the last reset request in a behavior. However, because the reset protocol can start in an arbitrary state, we relax this requirement and only ask that this property hold after a bounded amount of time. But a causal behavior should also ensure the flip side of the above condition: reset requests should result in signal events. A protocol that simply ignored reset requests would be useless. We specify the second condition by requiring that a reset request at a node u is followed in linear time by a signal event at node u.

Definition 7.3.6 *We say that a behavior β is causal if:*

1. *There is some constant c such that every signal event a_k that occurs at time greater than $\beta.start + c \cdot n$ is preceded by a request event a_j that occurs in linear time before a_k.*

2. *A* SIGNAL$_u$ *event occurs within linear time after any* REQUEST$_u$ *event.*

We are now ready to describe the behaviors that should be produced by a reset protocol.

Definition 7.3.7 *We define the reset problem RP to be the behaviors β that are timely, consistent and causal.*

The following lemma is useful because it tells us that every suffix of a behavior in RP is also in RP.

Lemma 7.3.8 *If a behavior β is in RP, then any t-suffix of behavior β is in RP.*

Proof: The proof consists mostly of looking at each property in the definition of RP and showing that if a suffix of β does not have the property then neither does β. The only tricky case is the property that all messages received after at most $O(n)$ time from the start of a behavior are normal. However, this can be deduced from the fact that the send event corresponding to a normal receive event in β occurs at most $O(n)$ time before the receive event. ∎

7.3.4 Alternative Specifications of Reset Problem

Traditional definitions (i.e., [AAG87]) of a reset service define the correctness of a reset service in terms of the states of protocol P, the user of the reset service. Our definition is more modular because it focuses on the behavior of the reset subsystem without any reference to the states of the users of the reset service.

Next, one might like a reset protocol to satisfy a much stronger consistency property than the one specified above. In the stronger condition, the mating relation between signal intervals would also be *transitive.*. For instance, suppose u, v and w are connected in a cycle such that u, v, and v, w, and w, u are all neighbors. Also suppose

that a signal interval S_u at u mates to a signal interval S_v at v, S_v mates to some S_w at w, and S_w mates to some interval S'_u at u. Then our consistency specification allows S_u to be different from S'_u. However, transitivity requires that $S_u = S'_u$. The stronger condition seems to capture the essence of network synchronization in that the signal events provide consistent time points across the entire network. However, we show in the appendix that our reset protocol (and the reset protocol of [AAG87]) *does not* satisfy the stronger condition. The applications in this thesis do not need the stronger condition.

Note that the weaker condition does imply that there is a transitive mating relation between final signal intervals at all nodes in the network.

Also the specification of the behaviors of a stabilizing reset protocol is complicated by anomalies that occur at the beginning of the behavior such as abnormal messages and signals that are not "caused" by any requests. Given this difficulty, we might be tempted to specify the reset problem using *suffixes* of the behaviors of a non-stabilizing reset protocol. This results in a more elegant definition. However, if we choose that definition, then we do not know any reasonably simple proof technique to show that a protocol stabilizes to behaviors that are *suffixes* of a specified set of behaviors.[2]

7.4 Overview of the solution

In this section, we will give an overview of a stabilizing reset protocol. Our solution basically consists of stabilizing the reset protocol of [AAG87] using the method of Chapter 5. In the first subsection, we describe a simple reset protocol that is not stabilizing. In the next subsection, the problems of the simple reset protocol are used to motivate the main ideas behind our reset protocol. Next, we give an overview of the code. Finally, we end this section with an example execution of our reset protocol. The next section contains a formal description of our reset protocol.

[2]Such proofs seem to involve showing that every initial state of an automaton A is a reachable state of another automaton B. But familiar inductive proof techniques (such as invariant arguments, progress metrics etc.) do not seem to suffice to show that one state is reachable from another state.

7.4.1 Problems with a Simple Reset Protocol

Amazingly, the consistency condition for normal reset behaviors can be guaranteed by the following *Simple Reset Protocol* (SRP). This is a non-stabilizing protocol. So assume that this protocol begins in a state where all queues and links do not contain any messages.

In the absence of reset requests, nodes are in the so called *Ready* state. In this state, any message sent from (say u to v) is queued by u in an outbound queue for the link. When the message arrives at v, it is stored in a buffer which we will call $buffer_v[u]$. Eventually, the message stored in $buffer_v[u]$ is delivered to the user at v. Thus in the absence of reset requests, sent messages are delivered in FIFO order. In the following two paragraph description of the protocol, when we say "node u" we mean "the reset protocol at node u".

When node u receives a reset request (REQUEST_u), the reset protocol at u sends an ABORT packet to all its neighbors and goes into a special *Abort* mode where it waits until it gets an ABORT packet from all its neighbors. It does so by setting a boolean flag $ack_u[v]$ for all neighbors v to indicate that it is expecting an ABORT packet from v. A node u that receives an ABORT packet in the *Ready* mode behaves in almost the same way as a node that receives a reset request – i.e., u sends an ABORT packet to all its neighbors. However, in this case u sets $ack_u[v]$ for all neighbors v *except* the neighbor v from which it received the ABORT packet. If u receives an ABORT packet from neighbor v and $ack_u[v] = true$ then u sets $ack_u[v] = false$. As soon as $ack_u[v] = false$ for all neighbors v, u returns to the *Ready* mode and performs a SIGNAL_u action.

The consistency condition is guaranteed by two additional rules. When an ABORT packet from u arrives at v, $buffer_v[u]$ is emptied. Second, no packet in $buffer_v[u]$ is delivered while v is in *Abort* mode and until v has performed any outstanding SIGNAL_v action. Intuitively, sending an ABORT packet on a link and waiting until another ABORT packet returns on the link effectively "flushes" any old messages that were sent in previous signal intervals. Essentially, we send an ABORT packet between any two SIGNAL events at a node and we delay delivery of packets until an outstanding signal has been performed. This ensures that a signal interval at u "communicates" or mates with at most one signal interval in v. The Simple Reset Protocol (SRP) is similar to the Chandy-Lamport snapshot protocol [CL85] with ABORT packets replacing "markers".

The problem with SRP is that it can easily be placed in a state where it never

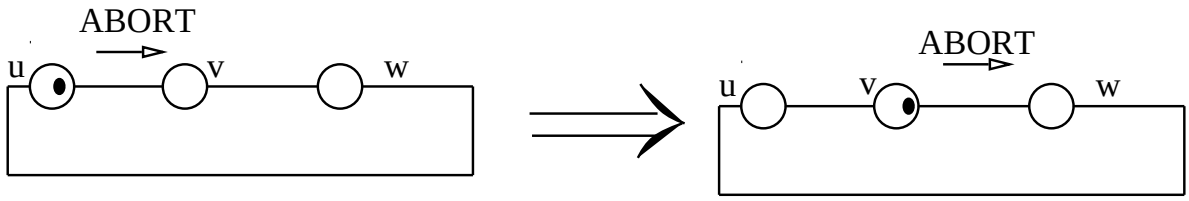

Figure 7.5: Two "bad" states of the Simple Reset protocol. The black dot before an edge indicates that a node is waiting for an ABORT on that edge.

terminates – i.e., violates the causality property. Consider the topology shown in Figure 7.5. Suppose in the initial state there is an ABORT packet in the channel from u to v, and u is waiting for an ABORT from v (but not from w) before u can exit from the *Abort* mode. Nodes v and w are in the *Ready* mode, and all other links are empty. Next, assume that the ABORT packet arrives at v which causes v to enter ABORT state and send an ABORT packet to u and w. Assume that the ABORT packet sent to u arrives quickly and causes u to return to *Ready* mode. Notice that by the rules of the protocol, v does not expect an ABORT from u. The resulting state is shown in Figure 7.5. By symmetry, it is clear that the execution can remain in a cycle of states where an ABORT packet continuously cycles through the network.

Now, since SRP is a non-stabilizing reset protocol, it may be possible to show that (after proper initialization) SRP will never enter such a "bad state". If the network is static, then this is indeed true. However, if network can have links that can fail and recover (a so called "dynamic network") then a series of link failures and recoveries can leave SRP in the "bad" state depicted in Figure 7.5.

More importantly for our purposes, SRP does not seem like a suitable point of departure for constructing a stabilizing reset protocol. Specifically, there does not appear any easy way to make SRP locally checkable. For instance, consider the topology of Figure 7.5. In a "good" state of SRP, if there is an ABORT packet in the cycle, there must be at least one other ABORT packet in the cycle, which is travelling in the "opposite" direction. This seems hard to check locally, even with the addition of a small amount of state. Instead, our point of departure is the AAG reset protocol of [AAG87]. This protocol works in dynamic networks and can be made locally checkable and correctable. We describe some more details of how the AAG protocol works and why it avoids the problems of the Simple Reset Protocol in the appendix. The ap-

pendix also contains a description of the changes we made to make the AAG protocol stabilizing. For the rest of this chapter, we will describe our reset protocol that is based on (but not identical to) the AAG protocol.

7.4.2 The basic idea behind the Reset protocol

Our reset protocol uses essentially the same idea as SRP for ensuring consistency – once again consistency is ensured by using ABORT packets to "flush" old packets (that were sent in a previous signal interval) from links. However, our protocol is much more conservative about allowing a node to to return to *Ready* mode.

A global (but approximate) summary is as follows. The protocol responds to reset requests in three phases. In the first phase, ABORT packets are sent out from nodes that receive reset requests. Nodes that receive ABORT packets and are in the *Ready* mode, broadcast ABORT packets to their neighbors. Thus the first phase consists of a wave of abort packets that spreads outwards from a reset request much the same way as in SRP. However, in our protocol the abort waves create an *abort forest* as they spread outwards. Consider any node u. Node u's parent in the abort forest is the neighbor from whom u last received an ABORT packet that caused u to leave *Ready* mode. If node u left *Ready* mode because u received a reset request, then u has no parent and u is considered a *root* in the abort forest. Notice that the abort forest is a set of *ad hoc* abort trees that are created while reset requests are being processed.

In the second phase, a node sends an ack to its parent when the subtree rooted at that node has stopped expanding. Thus, in the second phase a wave of acks flow up the abort trees to the roots. The first and second phases work in much the same way as the Dijkstra-Scholten termination detection algorithm [DS80]. As in the Dijkstra Scholten algorithm, some nodes may be in the first phase (forward propagation of ABORT packets) while other nodes may be in the second phase (sending acks up to parents).

What distinguishes our protocol from the Dijkstra-Scholten protocol is that there is a third phase in our protocol. When a root of an abort tree receives acks from all its children, it starts a wave of READY packets that flows down the tree and allows all nodes in the tree to return to the *Ready* mode. Thus the crucial difference between SRP and the AAG protocol is as follows. In the former, nodes return to *Ready* mode

after communicating with their neighbors. However, in our protocol a node u returns to *Ready* mode only after the root of the abort tree (that u is part of) knows that the abort wave has stopped propagating, and has informed u of this fact using a READY packet. Thus in the SRP protocol a node returns to *Ready* mode in constant time after it receives an ABORT packet. In our protocol a node may have to wait $O(n)$ time to return to *Ready* mode (potentially three worst case delays across the network, one for each phase).

The reader may wonder why three phases are needed. The appendix provides some intuition by describing why three phases were used in the original AAG protocol to avoid the problems of the Simple Reset Protocol.

7.4.3 Overview of the Code

The code of our reset protocol works as follows.

Each node u has three interesting variables. First each node has a mode, $mode_u$ which is one of *Ready*, *Abort* or *Converge*. *Ready*, is the "normal" mode of a node when it is not processing a reset request. If the mode of u is *Abort* or *Converge*, then this means that u is processing an abort request.

Next, each node u has an ack bit $ack_u[v]$ for each neighbor. If this bit is set, it indicates that u is waiting for an ack from v. (Unlike the Simple Reset Protocol, our protocol uses explicit ACK packets.) Finally, u has a parent variable $parent_u$ that points to the neighbor from which u received the reset request that it is processing. If u received a reset request through a REQUEST_u action (i.e., a reset request directly at u itself) then u sets $parent_u = nil$. If $parent_u = nil$, we will say that u is a root of an abort tree. A list of variables used by the code is shown in Figure 7.6.

In our code, the mode of a node is characterized by the state of the other variables at the node. Thus the mode of node u is really a derived variable:

$$mode(u) = \begin{cases} Abort, & \text{if } \exists v \text{ such that } ack_u[v] = true \\ Converge, & \text{if } parent_u \neq nil \text{ and } \forall v\,(ack_u[v] = false) \\ Ready, & \text{if } parent_u = nil \text{ and } \forall v\,(ack_u[v] = false) \end{cases}$$

Notice that unlike the Simple Reset Protocol, we have a third mode called *Converge*.

State

$ack_u[v]$: Boolean Flag for each neighbor v of u
$parent_u$: Either one of the neighbors of u or nil
$dist_u$: integer in the range $0..n'$, where n' is an upper bound on the number of nodes in the graph.
We also assume that for any (ABORT, d) packet, d is an integer in the range $0..n'$.
$signalbit_u$: Boolean flag (*used to remember to do a signal event*)
$free_u[v]$: Boolean Flag for each neighbor v of u (*says whether link to v is ready to accept a packet*)
$freem_u[v]$: Boolean Flag for each neighbor v of u (*says whether v is ready to accept a message*)
$queue_u[v]$: queue of size 5 consisting of packets drawn from P_{data} (*outbound queue for link to v *)
$buffer_u[v]$: queue of size 1 that can contain a Σ-message only (*inbound message queue for link from v *)

The following piece of code is used as a macro to propagate ABORT packets:

```
PROPAGATE_u(w, dist) ≡
  parent_u := w
  dist_u := dist
  For all neighbors v of u do                    (*broadcast abort packets to neighbors*)
    ack_u[v] := true
    enqueue (ABORT, dist + 1) on queue_u[v]
```

Figure 7.6: Variables and a Useful Macro

If $mode(u) = Converge$, this means that u has received acks from all its children and is waiting for a READY packet from its parent.

There are also three interesting packets used by the protocol: abort packets (that are encoded as by tuples of the form (ABORT, d) where d is a integer representing distance from the root), ack packets (that are encoded simply as ACK), and ready packets (that are encoded simply as READY).

The protocol will deadlock if the protocol is placed in an initial state in which the *parent* pointers form a cycle. In order to be able to locally check that the subgraph induced by the *parent* pointers is acyclic, we maintain a distance variable at each node, such that a node's distance is one greater than that of its parent. Specifically, distance is initialized to 0 upon a reset request, and its accumulated value is appended to any abort packets. Thus we encode an abort packet as a tuple (ABORT, d), where d is a distance.

The code that implements most of the reset protocol is shown in Figure 7.7. For convenience, we have marked certain transitions in the figure with the labels VR, VA, DA, IA, FA, RA and RR.

A VR (for *Valid Request*) transition is a reset request that causes a node to change its *mode* to *Abort*. A VA (for *Valid Abort*) transition is the receipt of an (ABORT, d) packet with valid distance field that causes a node to change its *mode* to *Abort*. A DA (for *Distance Invalid Abort*) transition is the receipt of an (ABORT, d) packet such that the distance field d is at the maximum value and such that its receipt causes a node to change its *mode* to *Abort*.

An IA (for *Invalid Abort*) transition is the receipt of an $(\text{ABORT}, *)$ packet that *does not* cause a node to change its *mode* to *Abort*. A FA (for *Final Ack*) transition is the receipt of an ACK packet that causes say node u to send an ACK packet to its parent. It is not hard to see that the ack that was received must have been the last ack that node u was waiting for. A RA (for *Root Ack*) transition is the receipt of an ACK packet at a root node that causes the root node to change its *mode* to *Ready*. A RR (for *Regular Ready*) transition is the receipt of an (READY) packet at a node that causes the node to change its *mode* to *Ready*.

Refer to these labels in Figure 7.7 when following the description below.

The code in Figure 7.7 uses a small macro called $\text{PROPAGATE}_u(v, d)$. This procedure is used to broadcast $(\text{ABORT}, *)$ packets and is shown in Figure 7.6. The first

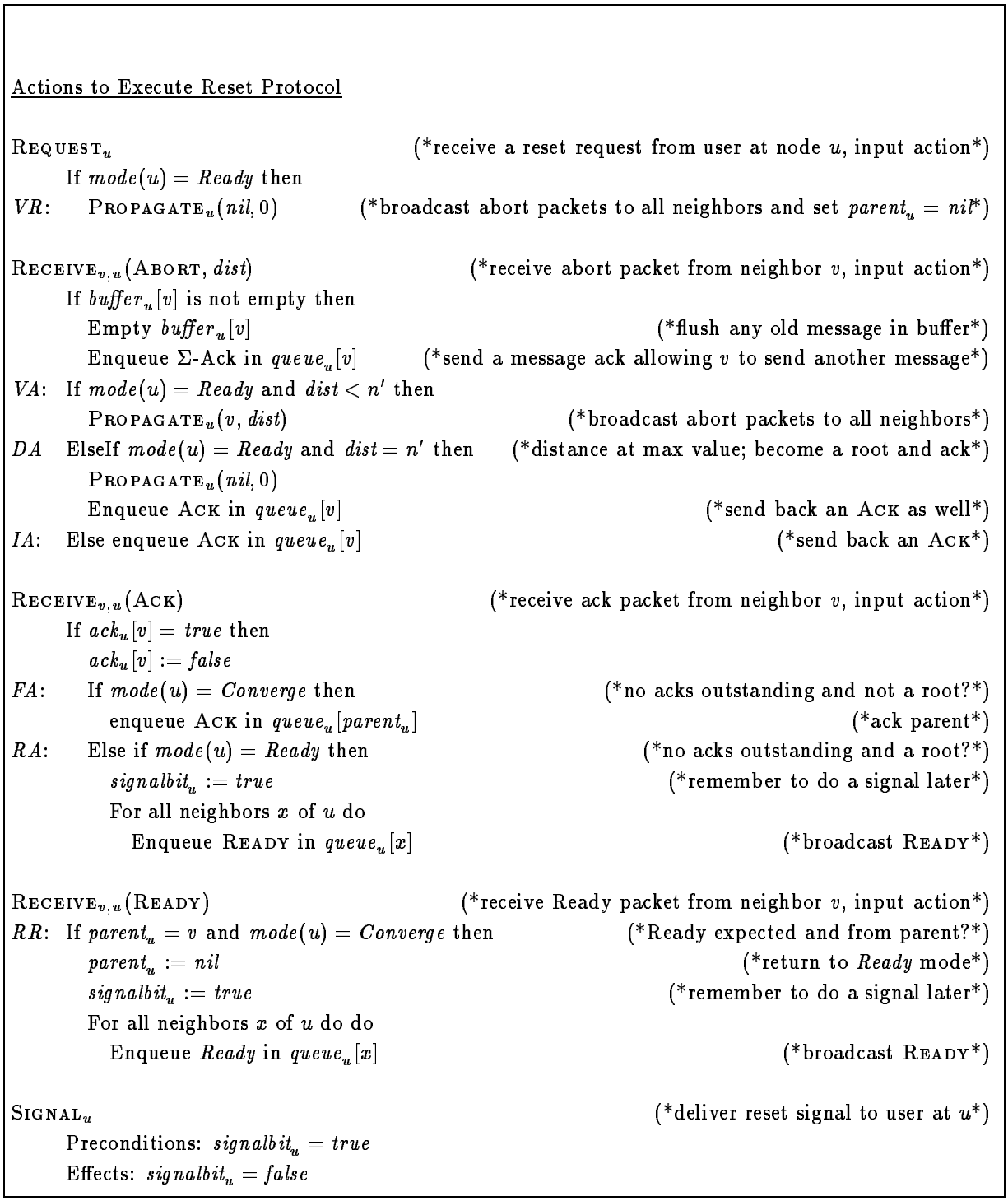

Figure 7.7: Actions at node u to execute reset protocol functions with respect to any neighbor v.

parameter to the procedure specifies that v is the new *parent* of u and the second parameter specifies that d is the distance from u to root of u's abort tree. The abort packets sent out as a result of this procedure (see Figure 7.6) will carry a distance value of $d + 1$. We add the distance variable to abort packets in order to make the protocol locally checkable.

When a reset request is made at some node u while in *Ready* mode (VR), u changes its mode to *Abort*, broadcasts an ABORT packet to all its neighbors, and sets its ack bits to *true* for all neighbors v. Node u then waits until all the neighboring nodes send back an ack packet. If node u receives an abort packet while in *Ready* mode (VA), it marks the neighbor from which the packet arrived as its parent, broadcasts ABORT, and waits for ACK packets to be received from all its neighbors. We also add a check to see whether the distance in the ABORT packet is less than n', which is the maximum value of the distance variable at a node. In linear time after all local predicates hold, this condition will always hold. However, this check helps to ensure that the local predicates remain closed during initial periods. If the distance check fails, (DA) node u becomes a root and sends out abort packets just as if it received a reset request; however, node u also sends back an ACK to the ABORT packet it received. Finally, if the ABORT packet is received by a node not in *Ready* mode (IA), an ACK is sent back immediately.

When node u receives an ACK from v it sets the ack bit for v to *false*. The action of node u when it receives the last anticipated ack depends on the value of *u's parent*. If u's parent is not *nil* (FA), u's mode is changed to *Converge*, and an ACK is sent to the parent. Notice that since $mode_u$ is a derived variable, $mode_u$ = *Converge* when $ack_u[v]$ = *false* for all neighbors v of u. If u is a root (RA), u changes its mode to *Ready* (by setting $parent_u$ = *nil*), and broadcasts a ready packet to all neighbors. If node u gets a READY packet from its parent while in *Converge* mode (RR), then u changes its mode to *Ready* (by setting $parent_u$ = *nil*), and broadcasts a ready packet to all neighbors.

Finally, whenever node u changes its mode from either *Abort* or *Converge* to *Ready* it sets a flag (which we call $signalbit_u$) to remind itself to later output a signal event. In other words, a SIGNAL_u event is enabled whenever $signalbit_u$ is set; naturally, when this event occurs, the flag is cleared. For convenience, we introduce the following definition.

Definition 7.4.1 *We say that node u has a* status *of* on *whenever $signalbit_u = false$ and $mode_u = Ready$, and* off *otherwise.*

The code that implements the sending and receiving of Σ-messages is shown in Figure 7.8.

If the status of u is on, u relays Σ-messages between the user and the network. More specifically, when a $\text{SENDM}_{u,v}(m)$ event occurs and node u has status on, u queues m on an outbound queue (called $queue_u[v]$ as in previous chapters) that contains packets to be sent to v. This queue can also contain an abort, ack or ready packet. Eventually, packets in this outbound queue are delivered to neighbor v. When v receives a Σ-message m from v, v queues m in an inbound message buffer called $buffer_v[u]$. Later, if v's status is on, it will do a $\text{RECEIVEM}_{u,v}(m)$ event and remove m from the buffer and deliver it to the user. See Figure 7.9 for a sketch of the inbound and outbound queues for a link.

If u's status is not on, u *discards* Σ messages input by the user through $\text{SENDM}_{u,v}$ actions. Also, u, will not do a $\text{RECEIVEM}_{v,u}$ action unless its status is on. Recall that all Σ-messages from a neighbor v are queued on $buffer_u[v]$. In order to separate packets that are sent during separate signal intervals at v, we use the same rule as the Simple Reset Protocol. *When a abort packet arrives from neighbor v, $buffer_u[v]$ is emptied.* Once again, this simple rule is really the key to ensuring the conditions required to satisfy the consistency property.

Note that $buffer_u[v]$ can store at most one message. Clearly, if user messages are not to be dropped, we have to rely on the FREEM action as a form of "flow control". Our scheme is as follows. We require that there is at most one Σ-message in transit from u to v. Thus u keeps a variable $freem_u[v]$. Whenever v delivers (or destroys) a Σ-message from u, v sends a Σ-Ack back to u. When u receives a Σ-Ack from v, u sets $freem_u[v]$ to *true*. This enables the $\text{FREEM}_u[v]$ action, which tells the user at u that it can safely send a message to v. Thus all we are doing is using a Σ-Ack message to provide feedback to the sender that the buffer at the other end is empty.

7.4.4 Example

Figure 7.10 describes a sample execution of the reset protocol for a simple topology consisting of four node u,v, w and x connected as shown in the figure. The figure

Actions to send and receive Σ-messages

SENDM$_{u,v}(m)$, $m \in \Sigma$ (*input action by which user sends a message*)
  If $mode(u) = Ready$ and $signalbit_u = false$ and $freem_u[v] = true$ then
  (* accept message only if mode is ready, no outstanding signals, and message flow control says OK*)
    enqueue m in $queue_u[v]$ (*enqueue m in outbound queue to v *)
    $freem_u[v] = false$ (*inhibit delivery of free event to user*)

SEND$_{u,v}(p)$ (*output action to send a packet on UDL link to v*)
  Preconditions:
    p is the head of $queue_u[v]$ (*p is head of outbound queue for link*)
    $free_u[v] = true$
  Effects:
    Delete p from $queue_u[v]$
    $free_u[v] := false$

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

RECEIVE$_{v,u}(m)$, $m \in \Sigma$ (*input action to receive a user message from v*)
  Enqueue m in $buffer_u[v]$ (*store m in inbound buffer for link*)

RECEIVE$_{v,u}(\Sigma$-Ack), (*input action to receive a Σ- message ack from v*)
  $freem_u[v] = true$ (*record that v is ready to accept a new message*)

RECEIVEM$_{v,u}(m)$, $m \in \Sigma$ (*output action to deliver a user message from v to the user*)
  Preconditions:
    m is the head of $buffer_u[v]$
    $mode(u) = Ready$ and $signalbit_u = false$ (*mode is ready and no outstanding signals*)
  Effects:
    Remove m from $buffer_u[v]$
    Enqueue Σ-Ack in $queue_u[v]$ (*send a message ack back to v*)

FREEM$_{u,v}$ (*output action to tell user that it can safely send another message to v*)
  Preconditions:
    $freem_u[v] = true$ and $signalbit_u = false$ and $mode(u) = Ready$
  Effects:
    None

Figure 7.8: Actions at a node u to send and receive user messages to and from any neighbor v of u.

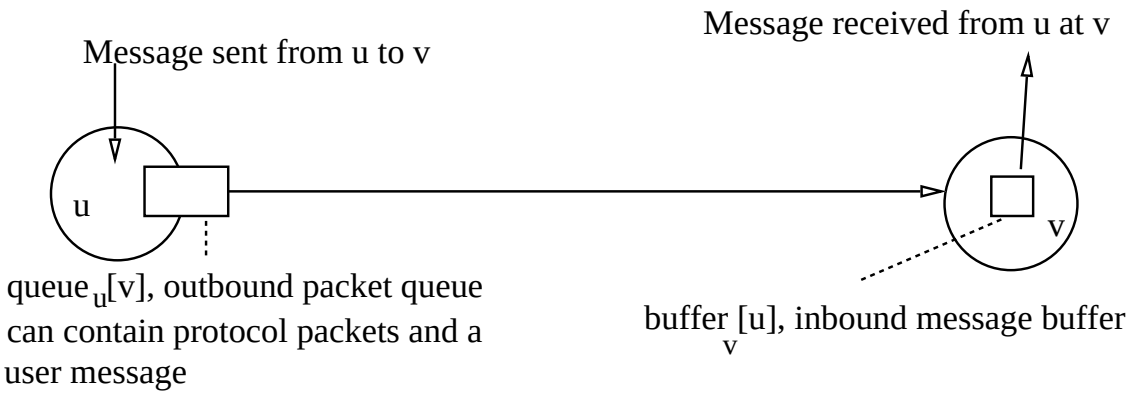

Figure 7.9: Sketch of the inbound and outbound queues for a link.

describes seven snapshots (F1-F7) taken during this sample execution.

The execution begins (F1) with nodes u and w receiving a snapshot request. Node u and w go into *Abort* mode and send abort packets to their neighbors. In F2, the abort packet from u has arrived at v. Next, u and w receive each other's abort packet and (F3) send back acks to each other. We assume that the abort from w to x travels slowly which allows the abort packet sent from v to arrive at x earlier (F3). Thus in F3, the abort tree rooted at u (which is shown using dotted lines) consists of u, v and x.

Next, x sends an abort to both v and w which are acked immediately. Then x receives the "slow" abort from w and sends back an ack. Once x gets an ack back from both v and w, x sends back an ack to its parent v (F4). When v receives this ack, it sends back an ack to its parent u (F5). Finally, in F6, u sends a ready packet down to v and then (F6), v sends a ready packet to x. Note that the ready packet destroys the abort tree as it travels down the tree.

7.5 Reset Automaton

In describing the reset code, we use the following notation. As in any node automaton there is a outbound queue of packets ($queue_u[v]$) at any node u for every neighbor v of u. As usual, $queue_u[v]$ consists of packet waiting to be sent on $C_{u,v}$.

Each node automaton N_u is specified given in Figures 7.6, 7.7, and 7.8. The code