- **Find a way of Compiling Stabilizing Protocols in the Shared Memory Model into our Network Model.** A number of interesting stabilizing protocols have been described using the low atomicity, shared memory model introduced by [DIM90]. It would be nice to have a compiler that that could convert protocols from their model to our network model and vice versa. This seems to be feasible.

10.2.2 Increased Understanding of Local Checking and Correction

Our understanding of local checking and correction is far from complete. This motivates the following problems:

- **Obtain a better understanding of local checkability.** Recall that a protocol is locally checkable for some good predicate L, if L is a conjunction of local predicates, and each local predicate is closed. Thus we can study this problem by asking two separate questions.
  - **How much state do we need to add to a protocol so that it's legal states are a conjunction of local predicates?** In the token passing protocol of Chapter 6 and the reset problem of Chapter 7, we made protocols locally checkable by adding a constant amount of state to each node. It is natural to ask what the minimum amount of storage is that must be added in order to make a protocol locally checkable.
  - **Can any weakly locally checkable protocol be transformed into an (equivalent) locally checkable protocol?** Recall that a weakly locally checkable protocol is a protocol whose legal states are a conjunction of local predicates; however, the local predicates are not necessarily closed. In Chapter 6 we described a simple protocol transformation that consisted of removing unexpected packet transitions. In many cases, this heuristic is sufficient to ensure that the local predicates are closed. We have used this heuristic successfully but we still don't completely understand when this heuristic is guaranteed to work. In Chapter 6, we described a sufficient condition called local extensibility. Are there weaker conditions than local extensibility?

- **Obtain a better understanding of local correctability.** In Chapter 5, we used a fairly operational definition of local correctability in terms of a local reset function. That definition is adequate for the thesis but it gives little insight into the structure of locally correctable protocols. Why are some protocols locally correctable but not others? Are there simpler sufficient conditions? It is also important to formally understand the connection between local correctability and locally checkable protocols that work in dynamic networks.

- **Generalize the Definitions of Locality and Local Checkability in Chapter 5**: In Chapter 5, we defined locality in terms of link subsystems. More generally, a subsystem is the composition of the nodes and channels corresponding to some subgraph of the network graph. For example if we define locality using the subsystems corresponding to the entire network graph, then our method reduces to the method of Katz and Perry [KP90]. Another interesting subsystem would consist of a single node and all its incoming and outgoing channels. Another interesting possibility is to consider subgraphs defined by the sparse network partitions [AP90] defined by Awerbuch and Peleg.

  Another simple generalization is to allow more than one local predicate per local subsystem. For example, consider an example consisting of two link subsystems and four link predicates, L_1, L_2, L_3 and L_4. Suppose the system is locally correctable using a partial order $<$ such that $L_1 < L_2 < L_3 < L_4$. Then as we apply local checking and correction to this system L_1 will become true first, followed by L_2 followed by L_3 and then L_4. Having multiple local predicates per local subsystem is only useful if these local predicates are independently ordered in the partial order. (If this were not the case we could simply work with a new local predicate that is the conjunction of all the predicates.) An example of the need for this generality is the *Rollback* protocol in Chapter 9.

- **Obtain a better understanding of Synchronous Checking** In Chapter 9, we introduced the notion of synchronous checking, which is a synchronous protocol that can check the output of another synchronous protocol. Synchronous checking is a little easier than asynchronous checking because all the nodes run in lockstep and we do not have to worry about messages in channels. Also, the checker need only check the *final output* of the protocol and not the intermediate states. Are there synchronous checkers for minimum spanning tree and max flow

protocols that are faster than the protocols being checked?

10.2.3 New Algorithms

We suggest the following algorithmic problems:

- **Invent a practical stabilizing reset protocol that stabilizes in $O(d)$ time, where d is the actual diameter of the network.** By way of comparison, the reset protocol of Chapter 7 stabilizes in $O(n)$ where n is the number of nodes in the network. This seems to be a major open problem. A solution to this problem would yield $O(d)$ stabilizing solutions for the spanning tree, topology update, and other problems described in Chapters 8 and 9. To be practical, the constants hidden in the $O(d)$ notation must be small (between 1 and 3).

- **Invent a practical stabilizing token passing protocol for rings and arbitrary graphs**: We have already described how to construct a tree from an arbitrary graph, and shown how to execute a stabilizing token passing protocol on a tree. However, the latency of token traversal on a tree can be quite high, for example if the original graph is a ring. The first example in [Dij74] appears extensible to token passing in a ring. An efficient stabilizing token passing protocol for rings would be useful and practical. Existing token ring protocols such as the IEEE 802.3 and FDDI protocols have a number of *ad hoc* mechanisms to deal with failures.

- **Invent a Stabilizing Clock Synchronization Scheme**: Fault-tolerant clock synchronization schemes typically have to defend against Byzantine faults. It would be desirable to invent a practical, stabilizing version of such a protocol. This would be an interesting example of a protocol that is robust against both Byzantine faults that continue as well as catastrophic faults that stop.

- **Simplify Existing Flow Control Schemes for Transport Protocols and Data Links**: A flow control scheme is a scheme by which a receiver regulates the rate at which a sender sends data in order to prevent buffer overflows at the receiver. In [CSV89] we propose an extremely simple stabilizing flow control scheme for physical links. It can be considered to be a trivial application of local checking and correction to the sender-receiver flow control protocol. The

resulting protocol is robust and simple enough to be implemented in hardware. Now, robust flow control schemes are a major component of Transport and Data link layer protocols. Perhaps existing flow control schemes can be simplified using local checking and correction.

- **Invent a stabilizing, distributed protocol to compute Sparse Partitions**: Awerbuch and Peleg [AP90] have shown how to decompose a network into what they call sparse partitions. They use sparse partitions to build efficient solutions for online tracking of mobile users, network synchronization, and network routing with low memory. The most efficient *distributed algorithm* for constructing sparse partitions is based on the work of Linial and Saks [LS91]. A stabilizing distributed algorithm for sparse partitions would be an extremely useful tool.

- **Make the Synchronizer Methodology practical for networks with links of different speeds**: The synchronizer methodology was introduced in [Awe85] and is extended to a stabilizing setting in Chapter 9. However, the method suffers from a severe drawback in networks in which links have varying delays. Essentially, it slows down all links to the speed of the slowest link. It may be possible to modify the methodology to avoid this drawback.

- **Invent a stabilizing version of the Bootstrap Protocol for End-to-End Communication**: Our previous work on stabilizing end-to-end communication [APV91b] has concentrated on producing a stabilizing version of the simple and elegant Slide protocol [AGR92]. However, the Bootstrap Protocol [AG91] is more efficient in some cases, and so a stabilizing version would be of theoretical interest. The current solutions are still too inefficient (for example in storage) for the end-to-end problem to have any practical application.

10.2.4 New Directions

We suggest the following new directions for research in self-stabilization.

- **Local Checking for Randomized Protocols**: Local checking of randomized protocols is an important research area. There are several randomized protocols

(e.g., the Rabin-Lehman dining philosophers protocol of [RL81]) that use randomization to break symmetry and to guarantee termination. Such protocols can often be locally checked in a deterministic fashion. For example, in graph coloring, it is easy to check whether a neighboring node has the same color. If, however, the check reveals a problem, a *randomized local reset action must often be applied.* Thus in graph coloring, a node may randomly choose a new color once it discovers that it has the same color as one of its neighbors. Now the randomized local reset functions can be applied at a node after all other nodes have already chosen their colors. Thus the dynamics (and the analysis) of the stabilizing protocol may be quite different from that of the original randomized protocol. Preliminary work in this area has been done by Baruch Awerbuch, Leonore Cowen, and Mark Smith at MIT.

Protocols whose behavior is correct only with high probability are even harder to check locally. Suppose that a (non-stabilizing) randomized protocol computes sparse partitions such that with high probability the partitions have low diameter. Then even if local checking can detect "poor" partitions, how can we tell whether the partitions are bad a) because of errors in the initial random bits at the nodes or b) because of a low probability outcome of the protocol?

- **Stabilizing Data Structures**: We have suggested earlier that programs that run on a single shared memory can be made stabilizing by using domain restriction – the states of the program are restricted to legal states. Consider the problem of building a stabilizing data structure (e.g., a dictionary or a queue) within the memory of a single processor. The data structure provides a certain set of operations (for example to insert and delete elements). The correctness of the data structure can be defined in terms of allowable sequences of operations. This notion of correctness is similar to the behavior specifications of I/O automata. Thus we can define a stabilizing data structure to be one in which the data structure can begin in an arbitrary state; however, any sequence of operations on the structure will eventually have a suffix that is a correct sequence (or a suffix of a correct sequence) of the data structure. For example, in a stabilizing dictionary, we would require that any elements inserted after the structure stabilizes would be found when searched for.

  Now any such data structure can be trivially made stabilizing in the following way. Before any operation on the data structure, we first check the invariants of

the data structure, and reinitialize the data structure if an error is found. Unfortunately, this slows down regular operations. Thus if we charge for processing (which we have not done in the distributed model in this thesis), there appears to be a tradeoff between stabilization time and the time to complete normal operations. For some data structures, the tradeoff can be extremely good. For example, we can easily implement a stabilizing, bounded size queue using a fixed size array. The head and tail pointers can be restricted to stay within the array bounds. However, it appears to be much harder to obtain good tradeoffs for say a tree-based implementation of a dictionary. We have done some preliminary thinking in this area [MPV90].

- **Applying Self-Stabilization to Other Areas**: Most of the stabilizing protocols described in this thesis are used for routing, scheduling, and resource allocation – tasks typically found in the network and Data Link layers of the communication hierarchy. However, in principle there is no reason why self-stabilization cannot be added as an "extra envelope" of fault-tolerance for higher layer protocols – e.g., file transfer and database protocols. Such protocols should be designed to avoid errors in the case of common faults such as node and link crashes. However, it is also desirable that these protocols recover by themselves after catastrophic errors. Self-stabilization is also applicable at the lowest layer of the protocol hierarchy. For example, it would be desirable to have stabilizing clock recovery and framing protocols at the physical layer.

- **Generalized Local Checking**: The method of Katz and Perry [KP90] consists of a single leader that checks the entire network. In our method of local checking, nodes independently (and in parallel) check each link subsystem. The Katz and Perry method is more general but less efficient. Local checking of link subsystems is efficient but only applicable to locally checkable protocols. Thus there may be intermediate approaches in which the notion of locality is generalized to an arbitrary set of subnetworks of the original network.

10.3 Summing Up

Self-stabilization abstracts the ability of a protocol to tolerate catastrophic faults that stop. On the other hand, the cost of self-stabilization is often low. Thus self-

stabilization can provide a cheap way to improve the fault-tolerance of network protocols.

Local checking can be used to *design* efficient, stabilizing protocols. The resulting protocols can be *proved* correct in a systematic way. The overhead of local checking can be *piggybacked* on existing keep-alive traffic between network nodes. Local checking may prove to be a useful *debugging* tool because it provides a continuous check of system predicates, and violations can be logged.

Our research into stabilization and local checking has helped us better understand existing protocols, both stabilizing and non-stabilizing. For instance, in the process of understanding Global Correction, we realized that a Global Reset protocol provides a mating relation that is a generalization of the guarantees of Data Link protocols. We also saw that many existing protocols use a special form of local checking that we called one-way checking.

Whitehead once said that the interest of a generalization is the interest of a road for those who know what travel is; and the pleasure of the road has its roots in the labor of the journey. After struggling to understand many examples of stabilizing protocols, we have begun to appreciate the simplicity of understanding provided by such concepts as local checking and counter flushing. They have helped us to see things a little more clearly and to travel a little distance. We hope that our readers will go much further.

Appendix A

Notation

We summarize notation that is commonly used in this thesis. Other definitions can be found using the index. For example, if the notation for $C_{u,v}$ refers to a UDL, the definition of a UDL can be found by using the index.

- $\{u, v\}$: For an pair of neighboring nodes u and v, this denotes the unordered pair corresponding to the directed edge (u, v). This is useful in defining a partial order on unordered pairs of nodes.
- $A|L$: For any automaton A and any subset L of the states of A, $A|L$ is the automaton that is identical to A except that the set of start states of $A|L$ is L.
- $C_{u,v}$: The Unit Capacity Data Link (UDL) corresponding to directed edge (u, v) in a topology graph.
- $Conj(\mathcal{L})$: For any link predicate set $\mathcal{L}$, $Conj(\mathcal{L})$ denotes the conjunction of the predicates in $\mathcal{L}$.
- $\mathcal{N}(t)$: For any network automaton $\mathcal{N}$, $\mathcal{N}(t)$ is the automaton that is identical to $\mathcal{N}$ except that the link and node delays in $\mathcal{N}(t)$ are equal to t.
- $Q_{u,v}$: The packet stored on the UDL corresponding to directed edge (u, v) in a topology graph. If the value of the variable is *nil*, then there is no packet stored.
- $queue_u[v]$: A queue of packets for outgoing edge (u, v) in the node automaton corresponding to node u. See the definition of a node automaton.

- $xqueue_u[v]$: The concatenation of $Q_{u,v}$ and $queue_u[v]$. Only used in some proofs. Can be the read as the extended queue at node u corresponding to neighbor v,
- $U(A)$: For any automaton A, $U(A)$ is the automaton that is identical to A except that the set of start states of $U(A)$ is equal to the set of states of A. Thus $U(A)$ creates a UIOA from an IOA.

The following symbols are used frequently to denote the following quantities

- $<$: A partial order on a set of local predicates.
- α: An execution of an automaton
- a_j: The j-th action in a behavior or execution.
- β: A behavior of an automaton.
- f: A reset function.
- G: A graph or a topology graph.
- L: A predicate (set of states) of an automaton.
- $L_{u,v}$ A local predicate for edge (u, v)
- $\mathcal{L}$: A set of local predicates.
- $\mathcal{N}$: A network automaton.
- $\mathcal{N}^+$: An augmented network automaton.
- P: A packet alphabet.
- s: A state of an automaton. A state with a subscript like s_i often denotes the i-th state in an execution.
- t: A time. Times with subscripts like t_n, t_p denote special constants; many of these are listed in the index.

Appendix B

Proofs for Chapter 5

Let $\mathcal{N}^+ = Augment(\mathcal{N}, \mathcal{L}, f)$. We will use s and $\tilde{s}$ to denote states of $\mathcal{N}^+$. In deriving time bounds recall that all locally controlled actions at nodes take t_n time; also for any link all actions have upper bound t_l.

B.1 Any Execution of $\mathcal{N}^+$ is infinite

Our first lemma states that all executions of $\mathcal{N}^+$ are infinite and hence in all executions of $\mathcal{N}^+$, time grows without bound. This follows because we have ruled out the possibility of so-called Zeno executions in our model (see Chapter 2). We will assume this implicitly in what follows without making explicit reference to this lemma.

Lemma B.1.1 *Any execution α of $\mathcal{N}^+$ is infinite.*

Proof: Suppose not. Then there is some final state s of α after which no action takes place. Consider any channel $C_{u,v}$. If $s.Q_{u,v} = nil$, then a $\text{FREE}_{u,v}$ action is enabled; but $s.Q_{u,v} = p \neq nil$, then a $\text{RECEIVE}_{u,v}(p)$ action is enabled. Both cases contradict the assumption that s is a final state. ∎

Notice also that since $\mathcal{N}^+$ is a UIOA, any suffix of an execution of $\mathcal{N}^+$ that begins with a timed state is also an execution of $\mathcal{N}$. We will assume this implicitly when we apply some of the lemmas and claims described below.

B.2 Basic Properties of links

The first lemma states that once a link is drop-free (see Definition 5.6.7), it remains drop-free.

Lemma B.2.1 *For any (u,v) and any transition $(s,\pi,\tilde{s})$ of $\mathcal{N}^+$, if $s \in F_{u,v}$, then $\tilde{s} \in F_{u,v}$.*

Proof: If $s.freeq_u[v] = true$ then since $s \in F_{u,v}$, $s.Q_{u,v} = nil$. Thus the only action that can cause $\tilde{s}.Q_{u,v} \neq nil$ is a $\text{SEND}_{u,v}(p)$ event that also sets $\tilde{s}.freeq_u[v] = false$. If $s.freeq_u[v] = false$ then the the only action that can cause $\tilde{s}.freeq_u[v] = true$ is a $\text{FREE}_{u,v}$ event which in turn can only occur if $s.Q_{u,v} = \tilde{s}.Q_{u,v} = nil$. ∎

The next lemma says that (u,v) becomes drop-free after the first action that sends a packet to $C_{u,v}$, or the first signal from $C_{u,v}$ that it is free.

Lemma B.2.2 *For any edge (u,v) and any execution α, (u,v) is drop-free in all states of α that follow a $\text{FREE}_{u,v}$ or a $\text{SEND}_{u,v}(p)$ action.*

Proof: After a $\text{FREE}_{u,v}$ action, $Q_{u,v} = nil$, and after a $\text{SEND}_{u,v}(p)$ action, $freeq_u[v] = false$. Hence if $\tilde{s}$ follows either of these actions in α, then $s_i \in F_{u,v}$. The lemma follows from the stability of $F_{u,v}$ (Lemma B.2.1). ∎

Once a link is drop-free we can be sure that all packets sent on the link will be delivered.

B.3 Time Bounds for Correction Phases

Recall the definition of a correction phase on a link from 5. This section builds up to a few important time bounds that are useful in the main body of the proof.

Only the last two lemmas are important. The remaining claims are only useful in proving the last two lemmas. In the following claims, when we say that an event occurs within time t in execution α, we mean that the event occurs within time t of the first state in α.

The first claim says that a packet on a link will be delivered in t_l time.

Claim B.3.1 *For all executions α and any (u,v), if in the initial state, $Q_{u,v} = p \neq nil$, then a* $\text{RECEIVE}_{u,v}(p)$ *event occurs within t_l time units.*

The next claim says that a sender will know that a link is free in $2t_l$ time. Also after this period, the link will be drop-free. (Intuitively it takes t_l time units to deliver any packet on the link, and t_l time units to deliver the FREE signal to the sender.)

Claim B.3.2 *For all executions α and any (u,v), within $2t_l$ time units there is a state s such that $s.freeq_u[v] = true$ and (u,v) is drop-free in s.*

Proof: By Claim B.3.1 in t_l time units after the first state of α, there must be a state s, such that $s.Q_{u,v} = nil$. Now we have two cases. First, suppose within t_l time units after s a $\text{SEND}_{u,v}(p)$ event occurs. Then in the state preceding this event, $freeq_u[v] = true$ and $Q_{u,v} = nil$ and we are done. On the other hand if the first case does not occur, then $Q_{u,v} = nil$ for t_l time units after s. Hence in t_l time units after s a $\text{FREE}_{u,v}$ action must occur, resulting in a state $\tilde{s}$ such that $\tilde{s}.freeq_u[v] = true$. By Claim B.2.2, (u,v) is drop-free in $\tilde{s}$. ∎

The next claim bounds the time before a response will be sent. Note that in the code, responses are sent continuously even without waiting for a request. This helps avoid deadlock in arbitrary initial states. We rely on the matching process to weed out meaningless responses.

Claim B.3.3 Response Time: *For all executions α and any leader edge (u,v), there must be some* $\text{SEND}_{v,u}(p_{resp})$ *action in α that occurs in $3t_n + 4t_l$ time units.*

Proof: From Claim B.3.2, within $2t_l$ time, there is a state s such that $s.freeq_v[u] = true$. If $queue_v[u]$ is empty for t_n time units after s, then a $\text{SEND}_{v,u}(p_{resp})$ will occur in this period. If not, $queue_v[u]$ becomes non-empty within time t_n after s, which enables the sending of a data packet. In this case, within $2t_n$ time units after s, a $\text{SEND}_{v,u}(p_{data})$ will occur, resulting in a state $\tilde{s}$ in which $turn_v[u] = response$. From Claim B.3.2, in 2 t_l time units after $\tilde{s}$, there is a state s' such that $s'.freeq_v[u] = true$ and $turn_v[u] = response$. Finally, in t_n time units after s', a $\text{SEND}_{v,u}(p_{resp})$ will occur. ∎

The next claim bounds the time before a new phase will start on a link.

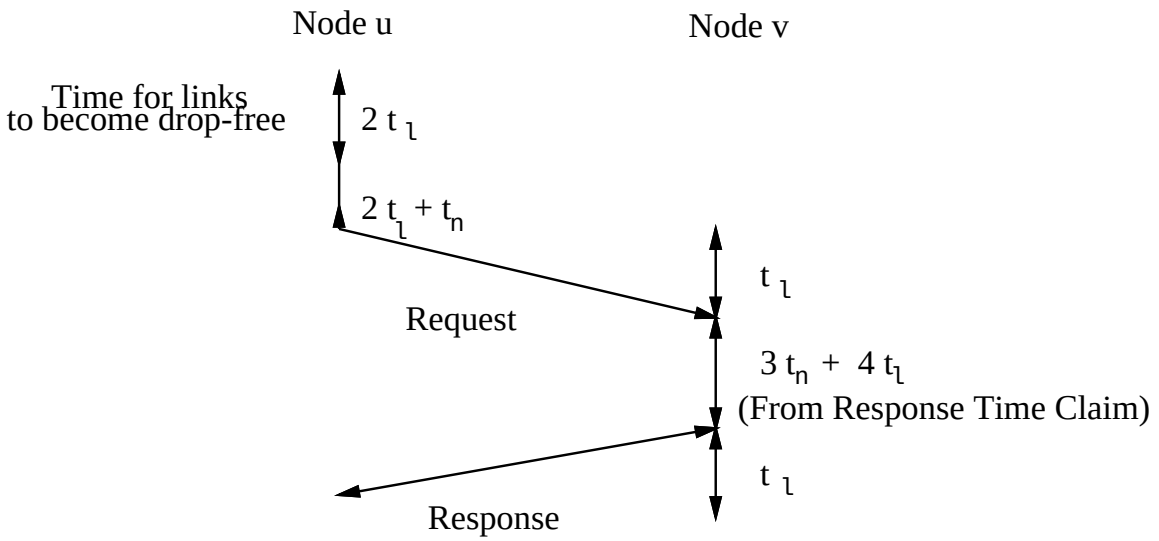

Figure B.1: Summary of the proof of Claim B.3.5.

Claim B.3.4 *For all executions α and any leader edge (u, v), within $2t_n + 2t_l$ time units there is a state in which $phase_u[v] = true$.*

Proof: Assume $phase_u[v] = false$ in the initial state of α or we are done trivially. From Claim B.3.2, within $2t_l$ time, there is a state s such that $s.freeq_u[v] = true$. If $queue_u[v]$ is empty for t_n time units after s, then a $\text{SEND}_{u,v}(p_{req})$ action occurs within this period, resulting in a state in which $phase_u[v] = true$. On the other hand, suppose that $queue_u[v]$ becomes non-empty within time t_n after s. Then a $\text{SEND}_{u,v}(p_{data})$ event will occur within $2t_n$ time units after s, resulting in a state in which $phase_u[v] = true$. ∎

The next claim bounds how long it can take for a phase on a link to complete once it has started.

Claim B.3.5 *For all α and any leader edge (u, v), within $4t_n + 10t_l$ time units there is a state in which $phase_u[v] = false$.*

Proof: The proof is summarized in Figure B.1. We assume the claim is false, so $phase_u[v] = true$ for $4t_n + 10t_l$ time units after the start of α. First, from Claim B.3.2 in $2t_l$ time, there is a state s such that both (u, v) and (v, u) are drop-free in s. Thus

any any requests and responses that are sent in states after s will be delivered to the corresponding receiver.

From Claim B.3.2, within $2t_l$ time after s, there is a state s' such that $s'.freeq_u[v] = true$. Thus a a $\text{SEND}_{u,v}(p_{req})$ action is enabled in s'. Thus within t_n time units after s' a $\text{SEND}_{u,v}(p_{req})$ action occurs. In t_l time units after this $\text{SEND}_{u,v}(p_{req})$ action, a $\text{RECEIVE}_{u,v}(p_{req})$ action occurs resulting in a state say s''. Also in all states after s'', $count_v[u] = count_u[v]$. Then by Claim B.3.3 in $3t_n + 4t_l$ after s'' a $\text{SEND}_{v,u}(p_{resp})$ event occurs with $p_{resp}.count = count_u[v]$. Finally, another t_l time units after this event, a $\text{RECEIVE}_{v,u}(p_{resp})$ event occurs with $p_{resp}.count = count_u[v]$. In the state immediately after this event, $phase_u[v] = false$. This contradicts the assumption that $phase_u[v] = false$ for $4t_n + 10t_l$ time units after the start of α. ∎

All the previous claims are only useful in proving Lemma 5.6.5 and Lemma 5.6.6. We now prove Lemma 5.6.6.

Proof: First assume that $l(u,v) = u$. Now we consider two cases. If $s_0.phase_u[v] = false$ then by Lemma B.3.4, within $2t_n + 2t_l$ time after s_0, there is a state s_i in which $s_i.check_u[v] = true$. However, the action before s_i must be a $\text{SEND}_{u,v}(p_{data})$ event. (The only other action that sets $check_u[v] = true$ is a $\text{SEND}_{u,v}(p_{req})$ action, but such an action is not enabled if $queue_u[v]$ is non-empty and $phase_u[v] = true$.) If $s_0.phase_u[v] = true$, then by Lemma B.3.5 within $4t_n + 10t_l$ after s_0 we reach a state in which $phase_u[v] = false$ and we are back to the previous case. Thus in both cases, a $\text{SEND}_{u,v}(p_{data})$ occurs within t_p after s_0.

Now suppose that $l(u,v) = v$. Once again we have two cases. Suppose $s_0.turn_u[v] = data$. From Claim B.3.2, within $2t_l$ time after s_0, there is a state s_i such that $s_i.freeq_v[u] = true$ and $s_i.turn_u[v] = data$. Thus within t_n time after s_i a $\text{SEND}_{u,v}(p_{data})$ event occurs. If $s_0.turn_u[v] \neq data$, then by Lemma B.3.3, we see that in at most $3t_n + 4t_l$ after s_i, a $\text{SEND}_{v,u}(p_{resp})$ event occurs that will result in state in which $turn_u[v] = data$. Then we are back to the first case. In both cases, a $\text{SEND}_{u,v}(p_{data})$ occurs within t_p after s_0. ∎

B.4 Proof that Clean Edges remain Clean

The following claim is useful in determining how and when the counter values stored in the links and the receiver can change. Recall that we used *undefined* when there is

no counter value stored on a link.

Claim B.4.1 *In any execution, and for any leader edge (u,v), the only action that can add a new element that is not undefined to countset(u,v) is a* $\text{SEND}_{u,v}(p_{req})$ *action.*

Proof: The only other actions that affect *countset*(u,v) are a $\text{RECEIVE}_{u,v}(p_{req})$, $\text{SEND}_{v,u}(p_{resp})$ and $\text{RECEIVE}_{v,u}(p_{resp})$, none of which can add any new values other than *undefined*. ∎

A nice property is that once an edge becomes clean, it remains clean. This is stated in Claim 5.6.11. We now prove Claim 5.6.11.

Proof: We assume that leader edge (u,v) is clean in s and we show that it remains clean in $\tilde{s}$. We will refer to the five predicates in the definition of a clean link by their numbers. The stability of the first predicate follows directly from Lemma B.2.1.

If $s.phase_u[v] = false$ and $\tilde{s}.check_u[v] = false$, then no $\text{SEND}_{u,v}(p_{req})$ action can occur which by Lemma B.4.1 is the only action that can change *countset*(u,v). Thus the second predicate holds and the remaining three hold trivially. If $s.phase_u[v] = false$ and $\tilde{s}.check_u[v] = true$, then π must be a $\text{SEND}_{u,v}(p_{req})$ or a $\text{SEND}_{u,v}(p_{data})$ event which will cause Predicate 3 to hold. Also by Predicate 2 in state s, $s.count_u[v] \neq s.respcount(u,v)$ and $s.count_u[v] \neq s.count_v[u]$. Since a $\text{SEND}_{u,v}(p_{req})$ or a $\text{SEND}_{u,v}(p_{data})$ event does not change $count_u[v]$ or *respcount*(u,v) or $count_v[u]$, Predicates 4 and 5 hold in $\tilde{s}$; also Predicate 2 holds trivially.

If $s.phase_u[v] = true$ and $\tilde{s}.check_u[v] = false$, then π must be a $\text{RECEIVE}_{v,u}(p_{resp})$ action and $s.count_u[v] = respcount(u,v)$ and $\tilde{s}.count_u[v] = (s.count_u[v] + 1) \bmod 4$. Then we can use the fact that Predicates 3 and 4 hold in s to infer that Predicate 2 holds in $\tilde{s}$. Also Predicates 3, 4, and 5 hold trivially in $\tilde{s}$. If $s.phase_u[v] = true$ and $\tilde{s}.check_u[v] = true$, then the only actions of interest are $\text{SEND}_{u,v}(p_{req})$ (which makes Predicate 3 true and leaves the others true), $\text{SEND}_{v,u}(p_{resp})$ (which makes Predicate 4 true and leaves the others true) and a $\text{RECEIVE}_{u,v}(p_{req})$. For a $\text{RECEIVE}_{u,v}(p_{req})$, we can use the fact that Predicate 3 holds in s to infer that Predicate 4 holds in $\tilde{s}$. All other Predicates hold trivially. Note that π cannot be a $\text{SEND}_{u,v}(p_{data})$ event because $s.phase_u[v] = true$. ∎

B.5 How Links Become Quiet: Detailed Proofs

Recall the definition of a quietlink from Chapter 5. For (u,v) to be quiet, we want to show not only that $L_{u,v}$ holds by the 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 reset transition.

Definition B.5.1 *Consider any transition $(s,\pi,\tilde{s})$ of $\mathcal{N}^+$. We say that this transition is a reset transition at node u with respect to node x if:*

- $s.mode_u[x] = reset$ *AND*
- *Either π is a* $\text{SEND}_{u,x}(p_{resp})$ *or π is a* $\text{RECEIVE}_{x,u}(p_{resp})$ *and* $l(u,x) = x$.

Intuitively, a reset transition at node u causes the local reset function f to be applied to the state of u.

We start by proving 5.6.17, which is the stability condition for a quiet link. **Proof:** We will show that each of the five predicates used to define a quiet link (Definition 5.6.15) holds in $\tilde{s}$. We will refer to the four predicates using the numbers given in Definition 5.6.15. The first predicate holds in $\tilde{s}$ because of Claim 5.6.11.

Consider the second predicate. We wish to show that $(\tilde{s}|u, \tilde{s}|(u,v), \tilde{s}|(v,u), \tilde{s}|v) \in L_{u,v}$. We know that the second predicate holds in s. We also know that if π is any action of $\mathcal{N}$, then the second predicate holds in $\tilde{s}$ as well because $L_{u,v}$ is a closed predicate by definition. The only other transitions of $\mathcal{N}^+$ that can affect $L_{u,v}$ are reset transitions at u or v.

Suppose $(s,\pi,\tilde{s})$ is a reset transition at u. From the fact that the third predicate holds in s, it is easy to see that $(s,\pi,\tilde{s})$ cannot be a reset transition at node u with respect to v. Suppose $(s,\pi,\tilde{s})$ is a reset transition at node u with respect to some neighbor $x \neq v$. Let e denote the leader edge corresponding to edge (u,x). Then e is not quiet in state s. From the hypothesis, we infer that $\{u,x\} \not< \{u,v\}$. Thus by the stability condition for local correctability, and since $\tilde{s}|u = f(s|u,x)$, we infer that the second predicate holds in $\tilde{s}$ even if $(s,\pi,\tilde{s})$ is a reset transition at u. A similar argument works if $(s,\pi,\tilde{s})$ is a reset transition at v.

We can infer that the third predicate holds in $\tilde{s}$ from the fact that the third, fourth and fifth predicates hold in s. The only actions to consider are $\text{RECEIVE}_{v,u}(p_{resp})$ and a $\text{RECEIVE}_{u,v}(p_{req})$. We can infer that the fourth predicate holds in $\tilde{s}$ from the fact that the third and fourth predicates hold in s. The only action to consider is a $\text{SEND}_{u,v}(p_{req})$ action.

Consider the fifth predicate. Let X be the predicate $Q_{u,v} = p_{resp}$ and $p_{resp}.count = count_u[v]$. The only actions that can change the truth or falsity of predicate X are $\text{SEND}_{v,u}(p_{resp})$ or $\text{RECEIVE}_{v,u}(p_{resp})$. Suppose that either π is a $\text{SEND}_{v,u}(p_{resp})$ action and $s.count_u[v] \neq s.count_v[u]$ OR π is a $\text{RECEIVE}_{v,u}(p_{resp})$ action. Then X will become false in $\tilde{s}$ and the fifth predicate holds trivially. On the other hand, suppose π is a $\text{SEND}_{v,u}(p_{resp})$ action and $s.count_u[v] = s.count_v[u]$ Thus X will become true in $\tilde{s}$. However, in this case, from the definition of a clean link, $s.Q_{u,v} \notin P_{data}$. Thus π will make the fifth predicate hold in $\tilde{s}$.

Thus, suppose that X is true in s and also in $\tilde{s}$. The only other actions to consider are actions that change the state of $s|u$. We know that if π is any action of $\mathcal{N}$, then the fifth predicate holds in $\tilde{s}$ as well because $L_{u,v}$ is a closed predicate by definition. The only other transition of $\mathcal{N}^+$ that can affect the fifth predicate is a reset transition at u. Suppose $(s, \pi, \tilde{s})$ is a reset transition at node u with respect to some neighbor $x \neq v$. Let e denote the leader edge corresponding to edge (u, x). Then e is not quiet in state s. From the hypothesis, we infer that $\{u, x\} \not< \{u, v\}$. Thus by the stability condition for local correctability, and since $\tilde{s}|u = f(s|u, x)$, we infer that the fifth predicate holds in $\tilde{s}$ even if $(s, \pi, \tilde{s})$ is a reset transition at u. ∎

Next we show 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.

In the rest of this section, we prove Lemma 5.6.18, which describes how and when links become quiet. A quick overview of this section can be obtained by skimming Claim B.5.4, Figure B.2, Definition B.5.7 and the last three lemmas in the section.

We start by proving some more detailed properties of the local snapshot and reset protocols during a clean phase. For all the claims and lemmas in this section, we fix an execution α of $\mathcal{N}^+$ and a leader edge (u, v). When we refer to a (u, v) phase we mean a (u, v) phase in α. When we refer to the code, we mean the code in Figures 5.6 and 5.7.

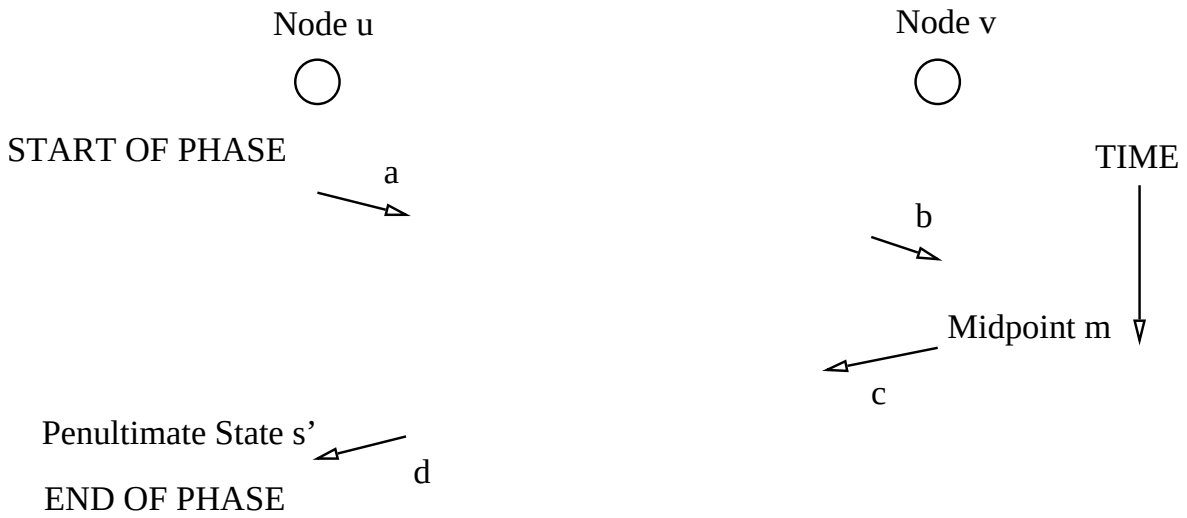

Figure B.2: More detailed structure of a clean phase

Since neither the mode or the counter at the leader can change except at the end of a phase it makes sense to talk of the mode and counter value of a phase.

Definition B.5.2 *Consider any clean (u,v) phase $\mathcal{P}$ with first state r. Then we use $mode(\mathcal{P})$ to denote $r.mode_u[v]$ and $count(\mathcal{P})$ to denote $r.count_u[v]$.*

Claim B.5.3 *For all states s in a clean (u,v) phase except the last state, $s.count_u[v] = count(\mathcal{P})$ and $s.mode_u[v] = mode(\mathcal{P})$.*

Proof: From the code, the values of $count_u[v]$ and $mode_u[v]$ can only change at the end of the phase. ∎

Figure B.2 shows the structure of a clean phase in more detail for some leader edge (u,v). The next claim formalizes the intuition behind Figure B.2 and defines two important states within a phase: the *midpoint* and the *penultimate state*.

Claim B.5.4 *Any clean (u,v) phase $\mathcal{P}$ contains the following four actions in the order shown:*

- *A* $\text{SEND}_{u,v}(p_{req})$ *action with $p_{req}.count = count(\mathcal{P})$ and $p_{req}.mode = mode(\mathcal{P})$.*

- *A* $\text{RECEIVE}_{u,v}(p_{req})$ *action with* $p_{req}.count = count(\mathcal{P})$ *and* $p_{req}.mode = mode(\mathcal{P})$.
- *Exactly one* $\text{SEND}_{v,u}(p_{resp})$ *action with* $p_{resp}.count = count(\mathcal{P})$ *and* $p_{resp}.node_state = m|v$, *where* m *is the state immediately following this action. We will call* m *the midpointphase!midpoint of the phase.*
- *Exactly one* $\text{RECEIVE}_{v,u}(p_{resp})$ *action with* $p_{resp}.count = count(\mathcal{P})$ *and* $p_{resp}.node_state = m|v$, *where* m *is the midpoint of the phase. We will call the state immediately before this action, the penultimate statephase!penultimate state of the phase.*

Proof: The four actions are shown in Figure B.2 as a, b, c, and d respectively. The midpoint and penultimate states are also marked.

Informally, the claim follows from two facts. First since both (u,v) and (v,u) are drop-free in all states of $\mathcal{P}$ any $\text{SEND}_{u,v}(p)$ action in $\mathcal{P}$ must be followed by a state in which $Q_{u,v} = p$. (i.e., any packet sent on channel $C_{u,v}$ will be stored in the link.). A similar statement holds for link (v,u). The second fact is that, by definition, at the start of a clean phase, $count_u[v] \notin countset(u,v)$. Thus at the start of the phase, there are no potentially confusing requests or responses numbered with the value of the counter at the sender. This ensures, for instance, that when a response p_{resp} is received at u with $p_{resp}.count = count_u[v]$, then p_{resp} was sent in the phase. Similarly it ensures that when a request p_{req} is received with $p_{req}.count = count_u[v]$, then the request was sent in the phase.

The formal argument is quite tedious. We start by proving the last statement in the claim and then working backwards to prove the other three. Let's sketch the first part of the argument.

We know that $\mathcal{P}$ can only end with a $\text{RECEIVE}_{v,u}(p_{resp})$ with $p_{resp}.count = count_u[v]$. Thus in the state s' before this action, $respcount(u,v) = count_u[v]$. But in the first state of $\mathcal{P}$, we know that since $\mathcal{P}$ is clean, $respcount(u,v) \neq count_u[v]$. Thus there must have been a $\text{SEND}_{v,u}(p_{resp})$ action in $\mathcal{P}$. Also, there can be only one such action in $\mathcal{P}$: any earlier $\text{SEND}_{v,u}(p_{resp})$ action would have caused the phase to have ended earlier; no later $\text{SEND}_{v,u}(p_{resp})$ action can occur because $Q_{v,u} \neq nil$ for all remaining states in the phase except the last state. If the state after the only $\text{SEND}_{v,u}(p_{resp})$ action is m, then from the code $p_{resp}.node_state = m|v$. Similar arguments can be used to show the remainder of the claim. ∎

In Figure B.2, in all states following b and before action c, $mode_v[u] = mode_u[v]$. Informally, this is because $mode_v[u]$ is set to the mode carried in the request packet and cannot change until action c occurs at which point $mode_v[u]$ is changed to *snapshot*. Then $mode_v[u]$ remains at this value till the end of the phase. Formally:

Claim B.5.5 *Consider a clean (u,v) phase $\mathcal{P}$ with midpoint m. Then in the state before m, $mode_v[u] = mode_u[v]$ and in all remaining states in $\mathcal{P}$, $mode_v[u] = snapshot$.*

Proof: After the first $\text{RECEIVE}_{u,v}(p_{req})$ action in $\mathcal{P}$, $mode_v[u]$ becomes equal to $mode_u[v]$. The value of $mode_v[u]$ can only change due to $\text{RECEIVE}_{u,v}(p_{req})$ actions and $\text{SEND}_{v,u}(p_{resp})$ actions. However, future $\text{RECEIVE}_{u,v}(p_{req})$ actions in $\mathcal{P}$ cannot change $mode_v[u]$ because from the definition of a clean link, $p_{req}.count = count_v[u]$. After the first (and only, see Claim B.5.4) $\text{RECEIVE}_{v,u}(p_{resp})$ action, $mode_v[u] = snapshot$ and remains at that value till the end of the phase. ∎

The following definition is convenient:

Definition B.5.6 *The leader edge corresponding to an edge (u,v) is (u,v) if $l(u,v) = u$ and (v,u) if $l(v,u) = v$.*

In order to guarantee correction or checking at end of a (u,v) phase, we need restrictions on the states of links adjacent to either u or v. Otherwise, concurrent checking/correction on these adjacent links may invalidate the checking/correction done in the (u,v) phase.

Definition B.5.7 *We will call a (u,v) phase $\mathcal{P}$* well-behaved *if $\mathcal{P}$ is clean and for all states s in the phase and for all $\{w,x\} < \{u,v\}$, the leader edge corresponding to $\{w,x\}$ is quiet in s.*

Now in a well-behaved phase, the response contains the state of v at the midpoint. However, the response is received in the penultimate state. Despite the fact that the state of v recorded in the response is "old", the recorded state is stil useful in the following precise sense.

Claim B.5.8 Phase Invariant: *Consider any clean (u,v) phase $\mathcal{P}$ with midpoint m and penultimate state s'. Then for all states r in the interval $[m,s']$, the following predicates hold in r:*

- $r.Q_{v,u} = p_{resp}$ *and* $p_{resp}.count = count(\mathcal{P})$.
- *For any* y*: If* $(y, nil, nil, p_{resp}.node_state) \in L_{u,v}$ *then* $(y, nil, nil, r|v) \in L_{u,v}$

Proof: The first part of the claim follows directly from Claim B.5.4. We establish the second predicate by induction on the length of the interval $[m, s']$. First, this predicate is true in m because $m.Q_{v,u} = p_{resp}$ and $p_{resp}.node_state = m|v$ by Claim B.5.4.

Next, assume the second predicate holds in some state r' before r and consider the transition (r', π, r). The only actions of interest are the actions that change the state of v. If π is any action of $\mathcal{N}$, then the second predicate holds in r as well because $L_{u,v}$ is a closed predicate. The only other transitions of $\mathcal{N}^+$ that can affect the state of v are reset transitions at v. We first see that (r', π, r) cannot be a reset transition at node v with respect to u. Suppose (r', π, r) is a reset transition at node v with respect to some neighbor $x \neq u$. Let e denote the leader edge corresponding to $\{v, x\}$. Then e is not quiet in state r'. But since the phase is well-behaved, we infer that $\{v, x\} \not\leq \{u, v\}$. Thus by the stability condition for local correctability, and since $r|v = f(r'|v, x)$, we infer that the second predicate holds in r. ∎

We can now state two lemmas that explain why the local snapshot and reset procedures work correctly. Let us call a snapshot phase a well-behaved phase $\mathcal{P}$ such that $mode(\mathcal{P}) = snapshot$. Similarly a reset phase is a well-behaved phase $\mathcal{P}$ such that $mode(\mathcal{P}) = reset$. The first lemma states that if $L_{u,v}$ does not hold at the end of a snapshot (u, v) phase, then at the end of the phase $mode_u[v] = reset$. This ensures that the next (u, v) phase will be a reset phase.

Lemma B.5.9 *For any snapshot* (u, v) *phase* $\mathcal{P}$ *with last state* s *the following is true. If* $(s|u, s|(u, v), s|(v, u), s|v) \notin L_{u,v}$*, then* $s.mode_u[v] = reset$.

Proof: Let s' be the penultimate state immediately before s in $\mathcal{P}$. The action just before s is a $\text{RECEIVE}_{v,u}(p_{resp})$ action. Since s' is the penultimate state of a snapshot phase, $mode_u[v] = snapshot$ and $Q_{v,u} = p_{resp}$ and $p_{resp}.count = count_u[v]$ in s'.

Thus, from the code, $s|u = s'|u$ and $s|v = s'|v$ (i.e., the basic states of nodes u and v remain unchanged after the $\text{RECEIVE}_{v,u}(p_{resp})$ action.). Next, from the fact that (u, v) is clean in s', we deduce that $s|(u, v) = s'|(u, v) = nil$. Also, after a $\text{RECEIVE}_{v,u}(p_{resp})$ action $s|(v, u) = nil$.

Thus if $(s|u, s|(u,v), s|(v,u), s|v) \notin L_{u,v}$ then $(s'|u, nil, nil, s'|v) \notin L_{u,v}$. But by the phase invariant (Claim B.5.8) if $(s'|u, nil, nil, s'|v) \notin L_{u,v}$ then we can be sure that $(s'|u, nil, nil, p_{resp}.node_state) \notin L_{u,v}$. Thus if $(s|u, s|(u,v), s|(v,u), s|v) \notin L_{u,v}$ then $(s'|u, nil, nil, p_{resp}.node_state) \notin L_{u,v}$. Hence, from the code of the RECEIVE$_{v,u}(p_{resp})$ action it is easy to see that $s.mode_u[v] = reset$. ∎

The second lemma states that at the end of a (u,v) reset phase, (u,v) is quiet.

Lemma B.5.10 *For any reset (u,v) phase $\mathcal{P}$ with last state s, (u,v) is quiet in s.*

Proof: Let s' be the state immediately before s in $\mathcal{P}$. The action just before s must be a RECEIVE$_{v,u}(p_{resp})$ action. Since s' is the penultimate state of a reset phase, $mode_u[v] = reset$ and $Q_{v,u} = p_{resp}$ and $p_{resp}.count = count_u[v]$ in s'. Let m be the midpoint of $\mathcal{P}$ and m' be the state immediately before m in $\mathcal{P}$. By Claim B.5.5, $m'.mode_v[u] = m'.mode_u[v] = reset$, and hence $p_{resp}.node_state = f(m'|v,u)$.

Now from the correction property of a local reset function, $(f(s'|u,v), nil, nil, f(m'|v,u) \in L_{u,v}$. Since $p_{resp}.node_state = f(m'|v,u)$, $(f(s'|u,v), nil, nil, p_{resp}.node_state) \in L_{u,v}$. But by the phase invariant, Claim B.5.8, this implies that $(f(s'|u,v), nil, nil, s'|v) \in L_{u,v}$. But by similar arguments as in the proof of the previous lemma: $s|(u,v) = nil$, $s|(v,u) = nil$, and $s|v = s'|v$. However, since $s'.mode_u[v] = reset$, $s|u = f(s'|u)$. Together, these equations imply that $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$.

Next, $s.mode_u[v] = s.mode_v[u] = snapshot$ by Claim B.5.3 and Claim B.5.5. Also, since (u,v) is clean in state s', if $s.Q_{u,v} = p_{req}$, then $p_{req}.count = s.count_v[u]$. Now it is easy to verify now that all five predicates used in the definition of a quiet link (Definition 5.6.15) hold in state s. ∎

We are now ready to prove Lemma 5.6.18. The proof is almost immediate from the last two lemmas. We recall the statement of Lemma 5.6.18. If every leader edge $(w,x) < (u,v)$ is quiet in some state s_i of some execution α of $\mathcal{N}^+|C$, then (u,v) is quiet in some state that occurs within $3 \cdot t_p$ of s_i in α.

Proof: First from the hypothesis and Lemma 5.6.17, every (u,v) phase in α is well-behaved. Consider the first (u,v) phase $\mathcal{P}$ in α that starts after state s_i. If $mode(\mathcal{P}) = reset$, then we know from Lemma B.5.10 that at the end of $\mathcal{P}$, (u,v) is quiet. If $mode(\mathcal{P}) = snapshot$, then we know from Lemma B.5.9 that at the end of $\mathcal{P}$, $mode_u[v] = reset$. Consider the next (u,v) phase in α, say $\mathcal{P}'$. Since $mode_u[v]$ only changes at the

end of a phase, $mode(\mathcal{P}') = reset$. Thus by Lemma B.5.10, (u, v) is quiet at the end of $\mathcal{P}'$.

Hence there is some state s_j that occurs before the end of the second (u, v) phase that follows s_i in α and such that (u, v) is quiet in s_j. Thus by the phase rate lemma, Lemma 5.6.5, s_j must occur within $3t_p$ time units after s_i in α. ∎

Appendix C

The AAG reset protocol

In this chapter, we describe why three phases were used in the original AAG protocol [AAG87] and also describe the changes that were required to convert the AAG protocol into the reset protocol described in Chapter 7. We also show that the mating relation provided by the AAG protocol is not transitive.

C.1 Why three phases are used in the AAG protocol

The AAG reset protocol [AAG87] is much more conservative than the Simple Reset Protocol about allowing a node to to return to *Ready* mode.

The point of all the conservatism in the AAG protocol is as follows. The use of three phases ensures that if reset requests stop being made, all nodes will eventually return to *Ready* mode. The AAG protocol makes this guarantee even in dynamic networks. The additional rules the AAG protocol uses to work in dynamic networks are remarkably simple. Suppose a link from node u to node v fails. If node v is node u's parent (in the abort tree), node u takes over as the root of the abort tree; if node u is expecting an ack from node v, node u assumes it has got an ack from v. Finally, when a link from u to v comes up, nothing special is done!

Here is an intuitive explanation of why three phases are used in the AAG protocol.

Each execution α of the reset protocol can be used to induce *work intervals* at nodes. Let us call a work interval at a node u, a maximal subsequence of α during which node u is not in *Ready* mode. Thus from the point of view of node u, each execution α can be divided into work intervals followed by *Ready* intervals (intervals during which u is in *Ready* mode.) Now each work interval at node u can be considered to be "caused" by an ABORT packet from a neighbor v (in which case we say that the work interval at u has as its parent a work interval at v) or by a reset request (in which case, we say that the work interval at u has no parent). Thus, starting from an execution α, we can assign each work interval at each node to a tree of work intervals.

We claim that each tree of work intervals has a height of at most n, where n is the number of nodes in the graph. This follows if we can show that any work interval tree can contain at most one work interval from any given node. This brings us to the crucial observation. Consider any work interval tree T in α. Let I_r be the work interval corresponding to the root. *The use of three phases guarantees us that there is some state in I_r that is contained in all the work intervals contained in T.* This is the state in which r sends READY packets to all its children. But that implies that a given node u cannot have two distinct work intervals in T because these two disjoint work intervals do not share a common state. Hence the height of T is at most n.

Thus, each root interval can "cause" at most one work interval at any node u. But each root interval corresponds to either a reset request (or to a link failure in the case of dynamic networks). Thus the number of work intervals at a node is at most the number of reset requests (plus the number of topology changes in the case of dynamic networks). Thus if the number of reset requests (and topology changes) is finite, there will only be a finite set of work intervals at each node. Next, it is possible to show that each work interval terminates in $O(n)$ time by using induction on the height of the abort tree. We combine the last two observations to prove the causality property – in finite time after all reset requests (and topology changes) stop, all nodes return to *Ready* mode.

It is interesting to return to the simple reset protocol (SRP). Consider an execution α of SRP that begins in a "bad state" as shown in Figure 7.5. Suppose we construct work interval trees from execution α. The result is that we get a single work interval tree of infinite height. Each work interval ends in finite time but there are an infinite number of work intervals at each node!

C.2 Overview of the changes required for stabilization

In a non-stabilizing setting (e.g. [AAG87], buffers like $buffer_u[v]$ are modelled by unbounded queues. However, just as in the case of links, stabilizing reset protocols must use bounded size queues if they are to stabilize in bounded time. Our solution is for v to use a single buffer to store messages from u, and to require that v sends a special $\Sigma - Ack$ packet whenever it removes a message from the buffer. This special packet is not needed in the original AAG protocol that uses unbounded queues.

The first step in making this protocol stabilizing is to make it locally checkable. A clear problem with the AAG protocol is that it will deadlock if in the initial state some *parent* edges form a cycle. As in stabilizing spanning tree algorithms [AKY90, AG90], we mend this flaw by maintaining 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 reset request, and its accumulated value is appended to the abort packets. Thus we encode an abort packet as a tuple (ABORT, d), where d is a distance.

Next, we list all the local predicates that are necessary to ensure correct operation of the Reset protocol. Note that a significant advantage of our approach is that all we have to do is to prove that all local predicates eventually hold. Once we do that we can rely on the correctness of the original protocol. However, since a rigorous correctness argument for the original protocol did not exist (as far as we knew), we produced a correctness argument anyway.

To ensure that the local predicates are closed predicates, we use the heuristic of removing unexpected packet transitions. Recall from Chapter 6, that to do this we add checks before processing any packet arriving at the node. We check whether this packet could have possibly been sent when the link subsystem (on which the packet arrived) is in a good state. Two checks we have added (which were not needed in the AAG protocol) are: when an ACK arrives, a node checks whether the ACK expected before processing it; when a READY packet arrives, a node accepts it only if it is in *Converge* mode and the packet has come from the node's parent.

The use of the distance variable introduces a new problem. Since the distance field has a maximum value, say n', we have to consider the case of a node u that receives an (ABORT, n') packet from neighbor v. If u is in *Ready* mode, u cannot simply accept v

as its parent and set its own distance to be one greater than n' since n' is the maximum value. Instead, in our code, u pretends that it has received a reset request before the (ABORT, n') packet. Thus u will first become a root and send ABORT packets to all its neighbors; then it will send back an ack to v. We call this a spurious reset request action.

Since the new action we have added is just a combination of two existing actions in the original protocol, the new action preserves all local predicates and consistency conditions of the original protocol. However, it does slightly complicate the proof of termination (and hence of the causality condition). Clearly, if the reset protocol can keep producing such "spurious reset requests" the protocol may never terminate even if all real reset requests stop. Luckily, it is easy to show that within linear time after all local predicates of the original protocol hold, no (ABORT, n') packets can be received. Thus spurious reset requests stop in linear time, and the termination proof is only slightly more complicated.

Next we have to design a local correction action for links, that is taken when a violation of the predicates is detected. The main difficulty about designing a correcting strategy is making it local, i.,e., to ensure that when we correct a link we do not affect the correctness of any other link. An interesting heuristic for this purpose is to notice that protocols designed for dynamic networks (like [AAG87]) had to deal with link failures and recovery. Now when a link fails and then immediately recovers, the original protocol must have established the local predicates for the link that failed without affecting the correctness of the other links. Thus the local correction procedure we use is essentially identical to the combined code in [AAG87] that is invoked when a link fails and recovers. This seems to be a powerful heuristic in general.

C.3 Mating Relation is not Transitive

Consider the reset protocol described in Chapter 7 (or the protocol described in [AAG87]) when all local predicates hold. We use the scenario shown in Figure C.1 to show that the mating relation between signal intervals at neighboring nodes is not transitive. Thus the reset protocol may cause inconsistent states of the user protocol during initial signal intervals. However, the mating relation between final intervals is indeed transitive. The original paper [AAG87] showed an example in which the

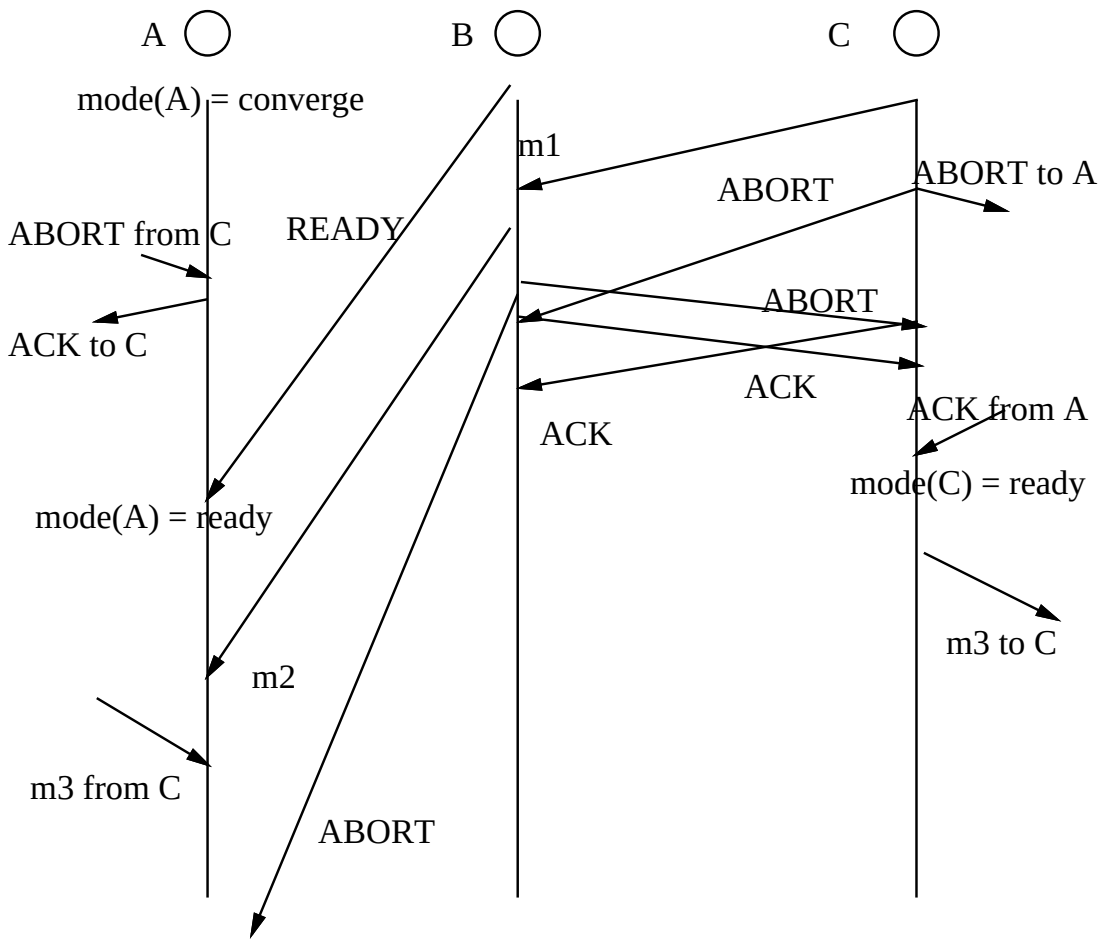

Figure C.1: Counterexample to Show that Mating Relation is not Transitive

mating relation was not transitive; however, the original paper used an optimization (in which ABORT packets were only sent on edges on which messages had either been sent or received in the last signal interval). Thus it was not clear whether the problem was due to the optimization. In fact, the simple counterexample in [AAG87] does not work as soon as we remove the optimization.

However, we show in Figure C.1 that there is a (more complicated) counterexample.

In Figure C.1 there are three nodes A, B and C that are connected in a cycle (not shown). The vertical axis represents time; time increases as we go downwards. Because this is a cycle, there is a link between A and C. When A sends a packet to C (or vice versa), we show this by showing an arrow leaving A and going off the left

of the page. We then depict the packet receipt at C by an arrow coming in to C from the right end of the page. Thus in the second event at C, C sends an ABORT packet to A which is received as the second event at A.

In the initial state, $mode(A) = Converge$ and $parent_A = B$. Thus A is waiting for a READY message from B that is sent out at the start of the execution. Assume that after B has sends out the READY packet, B does a SIGNAL event immediately and so $mode(B) = Ready$ at the start of the execution. We assume that $mode(C) = Ready$ and that there is no other packet in transit in the initial state. Clearly this is a valid initial state.

The first event at C is that C sends a user message (say $m1$) which is received by B and then delivered. Immediately after this B sends a user message $m2$ to A. Shortly after this, a reset request occurs at both B and C. This causes B to send an ABORT packet to A and C, and C to send an ABORT packet to A and B. The ABORT packet sent from C to A arrives before the READY packet arrives at A. Thus A will send an ACK back. Similarly B sends an ACK immediately back to C. The net result is that by the time A receives its READY packet, C is already in READY mode. We also assume C has done a signal event immediately after going to READY mode. Next the message $m2$ sent by B arrives at A and C sends a third user message $m3$ which is received by A. All this happens before the ABORT packet in transit from B arrives at A.

The net result is as follows. C sends two messages $m1$ and $m3$ in two different signal intervals at C, call them S_C and S'_C. Message $m1$ is received in a signal interval say S_B at B. Node B then sends a message $m2$ in signal interval S_B that is received in a signal interval (say S_A at A.) Finally the message sent by C in S'_C is also received in S_A.

By the definition of the mating relation, messages can only be received from a unique mate at a neighbor. Thus S_C mates to S_B and S_B mates to S_A. Also S'_C mates to S_A. If the mating relation were transitive, S_C would mate to S'_C which is impossible.

The significance of this counterexample is that arguments like "making a reset request guarantees a fresh start of the application protocol" do not work. If the application is doing any general form of checking, it may detect an inconsistent state and keep making reset requests, leading to non-termination. For example, if we were to use the global snapshot protocol due to [KP90] to check the application and then

use our stabilizing reset, the protocol may never terminate! Our stabilizing reset is still useful in a number of cases. For instance, (Chapter 8) when used in conjunction with a local snapshot protocol that checks for closed local predicates. In any case, termination of the resulting protocol requires careful argument.

Another significant thing about the counterexample is that it shows that our pairwise definition of a mating relation between neighbors is probably the only statement that one can make about the non-final intervals of a reset protocol. The original specification of [AAG87] used a state-based specification. It seems hard to specify this weak safety property of non-final intervals in terms of states as opposed to using an external behavior specification. In either, case [AAG87] only needed to specify the behavior in the final interval. For stabilizing applications, we *must* specify the behaviors in non-final intervals.

It is interesting to note that the reset protocol of Arora and Gouda [AG90] (after adaptation to a message passing model) is likely to ensure a transitive mating relation. This is because it does a reset protocol on a tree and only the root (effectively) sends out the quivalent of ABORT packets. Of course, our stabilizing protocol can be first used to construct a spanning tree, after which we do another reset which is guaranteed to work from a single root outwards.

Appendix D

Proofs for Reset Protocol in Chapter 7

D.1 Proving that the Local Predicates of the Reset Protocol are Closed

Recall the definition of $L_{u,v}$ in Definition 7.6.3.

Our strategy for showing that $L_{u,v}$ is closed is as follows. If we had to consider every action we would have a large number of cases to consider. However, each transition only affects a small number of variables. Thus we first isolate the transitions that can affect each variable. This reduces the number of cases we have to consider.

First, we need to define the key transitions. Recall that the code in Figure 7.7 has certain code paths marked as VR, VA, DA, IA, FA, RA, and RR. We now define these more precisely. It is helpful to refer to the code of Figure 7.7 in understanding the following definitions.

A informal description of these transitions is as follows. 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 $d < n'$ that causes a node to change its *mode* to *Abort*. A DA (for *Distance Invalid Abort*) transition is the receipt of an (ABORT, n') packet that causes a node to change its *mode* to *Abort*.

An *IA* (for *Invalid Abort*) transition is the receipt of an (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*. More carefully:

Definition D.1.1 *We call a transition* (s, a, s') *of* $\mathcal{R}$*:*

- *A VR transition at* u *if* $a = \text{REQUEST}_u$ *and* $s.mode(u) = Ready$.
- *A VA transition at* u *if* $a = \text{RECEIVE}_{*,u}(\text{ABORT}, d)$ *and* $d < n'$ *and* $s.mode(u) = Ready$.
- *A VA transition at* u *with respect to* v *if* $a = \text{RECEIVE}_{v,u}(\text{ABORT}, d)$ *and* $s.mode(u) = Ready$ *and* $d < n'$.
- *A DA transition at* u *with respect to* v *if* $a = \text{RECEIVE}_{v,u}(\text{ABORT}, d)$ *and* $s.mode(u) = Ready$ *and* $d = n'$.
- *An IA transition at* u *with respect to* v *if* $a = \text{RECEIVE}_{v,u}(\text{ABORT}, d)$ *and* $s.mode(u) \neq Ready$.
- *A FA transition at* u *with respect to* v *if* $a = \text{RECEIVE}_{*,u}(\text{ACK})$ *and* $s.mode(u) = Abort$ *and* $s.parent_u = v$ *and* $s'.mode(u) = Converge$.
- *A RA transition at* u *if* $a = \text{RECEIVE}_{*,u}(\text{ACK})$ *and* $s.mode(u) = Abort$ *and* $s'.mode(u) = Ready$.
- *A RR transition at* u *if* $a = \text{RECEIVE}_{*,u}(\text{READY})$ *and* $s.mode(u) \neq Ready$ *and* $s'.mode(u) = Ready$.

In the following lemma we will say that a Boolean condition b is *established* by some transition (s, a, s') if b is *false* in s but *true* in s'. Recall that we wish to prove that each $L_{u,v}$ is closed. The next lemma makes this job easier, by isolating the transitions that can establish various Boolean conditions used in the definition of $L_{u,v}$.

We start with the observation that we do not need to consider the transition DA explicitly because a DA transition at u with respect to v can be simulated by two other transitions: first a *VR* transition at u followed immediately by an *IA* transition at u with respect to v. Thus in the following lemmas and proofs we assume that the *DA* transition does not exist,

Lemma D.1.2

1. $ack_u[v]$ = *true can only be established by a VR or a VA transition at* u.
2. $A1(u,v)$ = *true can only be established by a VR or a VA transition at* u.
3. $A2(u,v)$ = *true can only be established by a VA transition at* v *with respect to* u.
4. $A3(u,v)$ = *true can only be established by a IA or a FA transition at* v *with respect to* u.
5. $A2(u,v)$ = *false can only be established by a FA transition at* v *with respect to* u.
6. $ack_u[v]$ = *false can only be established by a transition* (s,a,s') *such that* $s.ack_u[v]$ = *true and* $a = \text{RECEIVE}_{v,u}(\text{ACK})$.
7. $parent_u = v$ *can only be established by a VA transition at* u *with respect to* v.
8. $mode(u)$ = *Converge can only be established by a FA transition at* u *with respect to some neighbor* x.
9. $mode(u) \neq$ *Converge can only be established by a RR transition at* u.
10. $C1(u,v)$ = *true can only be established by a IA or a FA transition at* u *with respect to* v.
11. $C2(u,v)$ = *true can only be established by a transition* (s,a,s') *such that* $s.ack_v[u]$ = *true and* $a = \text{RECEIVE}_{u,v}(\text{ACK})$.
12. $C3(u,v)$ = *true can only be established by a RR or RA transition at* v.
13. $C2(u,v)$ = *false can only be established by a RR or RA transition at* v.

Proof: By inspection of the code. ∎

Notice that in the code we often enqueue packets to $queue_u[v]$. But since $queue_u[v]$ is finite (it only has room for 5 packets), this allows the possibility of a transition (s, π, s') causing a packet to be dropped if $queue_u[v]$ is full in the previous state. The next lemma shows that packets will not be dropped if $L_{u,v}$ holds in s. We will tacitly assume this lemma in what follows without making explicit reference to it.

Lemma D.1.3 *For any leader edge (u, v) and any transition (s, a, s') of $\mathcal{R}$, if $s \in L_{u,v}$. then in s', $\mathcal{Q}$ holds. Also if as part of the code for a, a packet p is enqueued on $queue_u[v]$ then p will be added to the tail of $queue_u[v]$ in s'.*

Proof: To show $\mathcal{Q}$, we show that when a packet of a certain type is in $xqueue_u[v]$, then no action will enqueue a packet of the same type. The five types of packets to consider are (ABORT, $*$) packets, ACK packets, READY packets, Σ – ACK packets and Σ-messages. The second part also follows from this claim because $xqueue_u[v]$ has room for five packets.

Suppose there is an (ABORT, $*$) packet in $xqueue_u[v]$ in s. Then (by $\mathcal{A}$) $s.mode(u) =$ *Abort* and hence this cannot be a *VA* or *VR* transitions at u. But these are the only transitions that can enqueue another (ABORT, $*$) packet to $xqueue_u[v]$ (see Lemma D.1.2, item 2).

Suppose there is an ACK packet in $xqueue_u[v]$ in s. Then, (by $\mathcal{B}$) $A1(v, u) =$ *false* and $A2(v, u) =$ *false* in s. Thus (by Lemma D.1.2, item 4), this cannot be a transition that can enqueue another ACK packet to $xqueue_u[v]$.

Suppose there is a READY packet in $xqueue_u[v]$ in s. Then (by $\mathcal{G}$) either $s.mode(u) =$ *Ready* or there is an ABORT in $xqueue_u[v]$ after the READY packet. In the former case, (by Lemma D.1.2, item 12) this this cannot be a transition that can enqueue another READY packet to $xqueue_u[v]$. In the latter case, (by $\mathcal{A}$) $s.ack_u[v] =$ *true*, and (by $\mathcal{B}$) $A3(u, v)$ is *false* in s'. Thus again (by Lemma D.1.2, item 12) this this cannot be a transition that can enqueue another READY packet to $xqueue_u[v]$.

Suppose there is a Σ message in $xqueue_u[v]$ in s. Then (by $\mathcal{H}$) in s, $freem_u[v] =$ *false*. Thus even if $a = \text{SENDM}_{u,v}(m)$, the code will not enqueue m on $xqueue_u[v]$.

Suppose there is a Σ – ACK message in $xqueue_u[v]$ in s. Then (by $\mathcal{H}$), $buffer_u[v]$ must be empty and so the code cannot queue a Σ – ACK on $xqueue_u[v]$. ∎

We now proceed to prove that each of the predicates from $\mathcal{A}$ to $\mathcal{H}$ are closed in a series of four lemmas: Lemma D.1.4, Lemma D.1.5, Lemma D.1.6, Lemma D.1.7, and Lemma D.1.8.

Lemma D.1.4 *For any leader edge (u,v) and any transition (s,a,s') of $\mathcal{R}$, if $s \in L_{u,v}$. then in s', $\mathcal{A}$ and $\mathcal{B}$ hold.*

Proof: We consider four cases:

- Suppose $ack_u[v] = \mathit{false}$ in s but $ack_u[v] = \mathit{true}$ in s'. Then (by $\mathcal{A}$) $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in s. Also by Lemma D.1.2, the transition must be a *VA* or a *VR* transition at u which causes $A1(u,v)$ to become *true* and leaves $A2(u,v)$ and $A3(u,v)$ as *false* in s'.

- Suppose $ack_u[v] = \mathit{false}$ in s and s'. Then (by $\mathcal{A}$) $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in s. Also by Lemma D.1.2, items 1 and 2, $A1(u,v)$ cannot hold in s' without making $ack_u[v] = \mathit{true}$ hold in s'. Also by Lemma D.1.2, item 3, $A2(u,v)$ cannot become *true* in s' if $A1(u,v)$ is *false* in s. Also by Lemma D.1.2, item 4, $A3(u,v)$ cannot become *true* in s' if $A1(u,v)$ and $A2(u,v)$ are *false* in s. Thus $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in s'.

- Suppose $ack_u[v] = \mathit{true}$ in s but $ack_u[v] = \mathit{false}$ in s'. By Lemma D.1.2, item 6, $a = \text{RECEIVE}_{v,u}(\text{ACK})$ and so $A3(u,v)$ is true in s. By $\mathcal{B}$, $A1(u,v)$ and $A2(u,v)$ are *false* in s. Also, (by $\mathcal{Q}$) there is exactly one ACK packet in $s.xqueue_v[u]$ and hence $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in s'.

- Suppose $ack_u[v] = \mathit{true}$ in s and s'. Then (by $\mathcal{A}$) exactly one of $A1(u,v)$,$A2(u,v)$, or $A3(u,v)$ is true in s. If $A1(u,v)$ is *true* in s, then $A1(u,v)$ becomes *false* in s' (by Lemma D.1.2, items 3 and 4) iff exactly one of $A2(u,v)$ or $A3(u,v)$ becomes *true* in s'. If $A2(u,v)$ is *true* in s, then $A2(u,v)$ becomes *false* in s' (by Lemma D.1.2, item 5) iff $A3(u,v)$ becomes *true* in s'. Finally if $A3(u,v)$ is *true* in s, then $A3(u,v)$ cannot become *false* in s' (by Lemma D.1.2, item 6) without causing $ack_u[v] = \mathit{false}$ in s', a contradiction.

∎

Lemma D.1.5 *For any leader edge (u,v) and any transition (s,a,s') of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{C}$ and $\mathcal{D}$ are true for (u,v) in s'.*

Proof: We consider cases:

- Suppose $parent_u \neq v$ in s'. Then $\mathcal{C}$ and $\mathcal{D}$ hold trivially in s'. Suppose $parent_u \neq v$ in s and $parent_u = v$ in s'. Then by Lemma D.1.2, item 7, this must be a VA transition at u with respect to v. Thus $s'.mode(u) = Abort$, Thus $A1(v,u)$ is *true* in s and hence (by $\mathcal{A}$) $A3(v,u) =$ *false* in s and hence $C1(u,v)$ is *false* in s and hence in s'. Also, (by $\mathcal{A}$) $ack_v[u] =$ *true* in s and s' and hence $C2(u,v)$ is *false* in s'. Next (by $\mathcal{G}$ and $\mathcal{Q}$) we can infer that $C3(u,v)$ is *false* in s and hence in s'. (This is because if $C3(u,v)$ were *true* in s then (by $\mathcal{G}$) there must be a second (ABORT, $*$) packet in $xqueue_v[u]$ which would violate $\mathcal{Q}$.)

  Thus in the remaining cases we assume that $parent_u = v$ in s and s'.

- Suppose $mode(u) \neq Converge$ in s and s'. Then $C1(u,v)$, $C2(u,v)$, and $C3(u,v)$ are *false* in s. Also by Lemma D.1.2, item 10, $C1(u,v)$ can become *true* in s' only if $mode(u) = Converge$ in s'. [1] By Lemma D.1.2, item 11, $C2(u,v)$ can become *true* in s' only if $C1(u,v)$ is true in s. Finally by Lemma D.1.2, item 12, $C3(u,v)$ can become *true* in s' only if $C1(u,v)$ or $C2(u,v)$ is true in s. Thus $C1(u,v)$, $C2(u,v)$, and $C3(u,v)$ are *false* in s'.

- Suppose $mode(u) \neq Converge$ in s and $mode(u) = Converge$ in s'. Then (by $\mathcal{C}$) $C1(u,v)$, $C2(u,v)$, and $C3(u,v)$ are *false* in s. Also by Lemma D.1.2, item 8, a is a FA transition at u with respect to v. Thus after this transition, $C1(u,v)$ becomes *true* in s', and $C2(u,v)$ and $C3(u,v)$ remain *false* in s'.

- Suppose $mode(u) = Converge$ in s and $mode(u) \neq Converge$ in s'. Then by Lemma D.1.2, item 9, a is a RR transition at u. Thus $C3(u,v)$ is *true* in s; hence (by $\mathcal{D}$) $C1(u,v)$ and $C2(u,v)$ are *false* in s and s'. Also, (by $\mathcal{Q}$) there is exactly one READY packet in $xqueue_v[u]$ in s and this is removed after the transition, and so $C3(u,v)$ is *false* in s'.

[1] Note that this cannot be an IA transition at u with respect to v because otherwise $s.mode(u) = Abort$ which together with $s.parent_u = v$ would violate $\mathcal{A}$.

- Suppose $mode(u) = Converge$ in s and s'. Then (by $\mathcal{C}$ and $\mathcal{D}$) exactly one of $C1(u,v)$,$C2(u,v)$, or $C3(u,v)$ is true in s. First suppose $C1(u,v)$ is *true* in s. Then (by $\mathcal{A}$) $ack_v[u] = true$ in s. Thus $C1(u,v)$ becomes *false* in s' iff exactly one of $C2(u,v)$ or $C3(u,v)$ becomes *true* in s'. Also if $C1(u,v)$ remains *true* in s', then $ack_v[u] = true$ in s' by Lemma D.1.2, item 6. Thus by Lemma D.1.2, items 11 and 12, $C2(u,v)$ and $C3(u,v)$ cannot become *true* in s'.

  Next, notice that since we have assumed that $mode(u) = Converge$ in s, $C1(u,v)$ remains *false* in s' (by Lemma D.1.2, item 10) if it is *false* in s. Suppose $C2(u,v)$ is *true* in s. Then $C2(u,v)$ becomes *false* in s' (by Lemma D.1.2, item 13) iff $C3(u,v)$ becomes *true* in s'.

  Finally if $C3(u,v)$ is *true* in s, then $C3(u,v)$ cannot become *false* in s' without causing $mode(u) = Ready$ in s', a contradiction. Also by Lemma D.1.2, item 11 $C2(u,v)$ cannot become *true* in s' since $C1(u,v)$ is *false* in s.

∎

Lemma D.1.6 *For any leader edge (u,v) and any transition (s,a,s') of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{E}$ and $\mathcal{F}$ are true for (u,v) in s'.*

Proof: First consider $\mathcal{F}$. Suppose there is no (ABORT, $*$) packet in $xqueue_u[v]$ in s. Then if an (ABORT, $*$) packet is in $xqueue_u[v]$ in s', then this must be a (by Lemma D.1.2, item 2) *VR* or *VA* transition at u. However, after such a transition there is an (ABORT, d) packet in $xqueue_u[v]$, where $d = dist_u + 1$. Suppose there is a (ABORT, d) packet in $xqueue_u[v]$ in s with $d = s.dist_u + 1$. Then (by $\mathcal{A}$) $s.mode(u) = Abort$. Now the only transitions that can change $dist_u$ are *VR* or *VA* transitions at u, but such transitions are not enabled in if $s.mode(u) \neq Ready$.

Now consider $\mathcal{E}$ and consider three cases.

- $parent_u \neq v$ in s': then $\mathcal{E}$ holds trivially.
- $parent_u \neq v$ in s but $parent_u = v$ in s': Then by Lemma D.1.2, item 7, a must be a RECEIVE$_{v,u}$(ABORT, d) event with $s.mode(u) = Ready$. But by $\mathcal{F}$ in s, $d = s.dist_v + 1$. Thus in s', $dist_u = dist_v + 1$. and $mode(u) \neq Ready$.

- $parent_u = v$ in s and s'. If in s, $C3(u,v)$ is *true*, then $C3(u,v)$ can become *false* in s' if $a = \text{RECEIVE}_{v,u}(\text{READY})$, but in that case $s'.parent_u = nil$. If in s, $dist_u = dist_v + 1$ and $mode(v) \neq Ready$, then this transition cannot be a VR or VA transition at v or u. But only such transitions can change either $dist_v$ or $dist_u$. Also if $s'.mode(v) = Ready$, then this must be a RR or RA transition at v which results in $C3(u,v)$ becoming true in s'.

∎

Lemma D.1.7 *For any leader edge (u,v) and any transition (s,a,s') of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{G}$ is true for (u,v) in s'.*

Proof: Suppose in s there is no p in $xqueue_u[v]$ such that p is either a READY packet or p is a Σ-message or p is a $\Sigma - \text{ACK}$. Then if there is such a packet in s', then, by the code, $s.mode(v) = Ready$ and we are done. Suppose in s there is a p in $xqueue_u[v]$ such that p is either a READY packet or p is a Σ-message or p is a $\Sigma - \text{ACK}$. Then by $\mathcal{G}$ either there is an $(\text{ABORT}, *)$ packet in $xqueue_u[v]$ or $s.mode(u) = Ready$. Consider the first case. Since the channel from u to v is FIFO, the $(\text{ABORT}, *)$ packet cannot be removed from $xqueue_u[v]$ in s' without also removing packet p. Consider the second case. Then if $s'.mode(v) \neq Ready$ then this transition must be a VA or VR transition at u which would result in adding an $(\text{ABORT}, *)$ packet to the end of $xqueue_u[v]$ in s'.
∎

Lemma D.1.8 *For any leader edge (u,v) and any transition (s,a,s') of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{H}$ is true for (u,v) in s'.*

Proof: We consider all the actions a that can affect this predicate. For each action considered, a symmetrical argument holds for the action with u and v interchanged.

If $a = \text{SENDM}_{u,v}(m)$ and $s.freem_u[v] = false$, then message m is dropped and there is no change to the concerned variables. If $a = \text{SENDM}_{u,v}(m)$ and $s.freem_u[v] = true$, then there is no Σ-message in $M_{u,v}$ in s and no $\Sigma - \text{ACK}$ in $s.xqueue_v[u]$. Also, (by $\mathcal{Q}$and $\mathcal{H}$) there is at most one $(\text{ABORT}, *)$, ACK, or READY packet in $queue_u[v]$ in s. Since $queue_u[v]$ can store five packets, after this event m is placed in $queue_u[v]$ and $s'.freem_u[v] = false$ and and there is no $\Sigma - \text{ACK}$ in $s.xqueue_v[u]$.

If $a = \text{RECEIVE}_{u,v}(m)$, $m \in \Sigma$, then m is enqueued in $buffer_v[u]$ and none of the concerned variables change. Note that by $\mathcal{H}$, $buffer_v[u]$ is empty in s. If $a = \text{RECEIVE}_{u,v}(\text{ABORT}, *)$ and $s.buffer_v[u]$ is empty in s, then there is no change to the concerned variables. Suppose $a = \text{RECEIVE}_{u,v}(\text{ABORT}, *)$ and $s.buffer_v[u]$ contains a message m in s or $a = \text{RECEIVEM}_{u,v}(m)$. Then (by $\mathcal{H}$) in s, $freem_u[v] = false$, there is no $\Sigma - \text{ACK}$ in $xqueue_v[u]$, and no other message in $M_{u,v}$ besides m in $buffer_v[u]$. Then in s', all variables remain unchanged except that $buffer_v[u]$ becomes empty and a $\Sigma - \text{ACK}$ is added to $s.xqueue_v[u]$.

Similarly, if $a = \text{RECEIVE}_{v,u}(\Sigma - \text{ACK})$, then by $\mathcal{H}$in s, $freem_u[v] = false$, there is no Σ message in $M_{u,v}$ and exactly one $\Sigma - \text{ACK}$ in $xqueue_v[u]$. The result is that the $\Sigma - \text{ACK}$ is removed from $xqueue_v[u]$ and $freem_u[v]$ becomes $true$ in s'. ∎

D.2 Proving that the Reset Protocol Behaviors are Timely, Causal, and Consistent

We will show that every behavior of $\mathcal{R}|L$ is timely and causal and satisfies the consistency property. We will do so in the next five subsections. First, we prove a series of useful preliminary lemmas. In the second subsection, we prove that every behavior is timely. In the third subsection, we prove that every behavior satisfies the consistency property. In the fourth subsection, we prove that every behavior is causal. Finally, we tie everything together and show that every behavior β of $\mathcal{R}|L$ is in RP.

We will assume this claim implicitly in what follows.

D.2.1 Useful Claims and Lemmas for Reset Protocol

The first lemma is the important Termination Lemma that states that the *mode* will become *Ready* in $O(n)$ time.

Lemma D.2.1 Termination Lemma *Consider an execution $\alpha = s_0, a_1, s_1, \ldots$ of $\mathcal{R}|L$. For any state s_i there is another state s_j that occurs within $O(n)$ time after s_i and such that $s_j.mode(u) = Ready$.*

Proof: A formal argument can be made based on the intuitive "proof" given in Section 7.7.1. We omit it here. ∎

The next lemma is the Signal Lemma. The lemma states that once the *status* of a node u is *off* (recall that this means that either $signalbit_u = true$ or $mode_u \neq Ready$) then a SIGNAL_u event is guaranteed to occur in linear time.

Lemma D.2.2 Signal Lemma: *For any execution α of $\mathcal{R}|L$, if $s_i.status(u) = off$ then a SIGNAL_u event occurs within $O(n)$ time after s_i.*

Proof: Suppose $s_i.mode(u) = Ready$ and $s_i.signalbit_u = true$. Then it is easy to see from the code that a SIGNAL_u action is enabled and will remain enabled until it occurs in constant time after s_i.

The only other possibility is that $s_i.mode(u) \neq Ready$. Let s_k be the first state after s_i such that $s_k.mode(u) = Ready$. We know from Lemma D.2.1 that such a state exists and s_k occurs within $O(n)$ time after s_i. Then $s_{k-1}.mode(u) \neq Ready$ and $s_k.mode(u) = Ready$. Thus from the code it is easy to see that $s_k.signalbit = true$. Now we are back to the first case and hence a SIGNAL_u event must occur in constant time after s_k. ∎

The next claim is a variation of the Signal Lemma. It states that the *status* of a node u cannot change from *off* to *on* until the next SIGNAL_u event.

Claim D.2.3 *For any execution α of $\mathcal{R}|L$, if $s_i.status(u) = off$ and $s_k.status = on$ and $k > i$, then there is a SIGNAL_u event between s_i and s_k in α.*

Proof: From the code, the only way $status(u)$ can change from *off* to *on* is by a SIGNAL_u event. Note that any action that changes $mode(u)$ to *Ready* also sets $signalbit_u$ to *true*, which leaves $status(u)$ unchanged. ∎

Next we show that any packet queued on the outbound queue for a link will be delivered in constant time. This follows from the fact that the outbound queue has a size of at most 4.

Claim D.2.4 *Consider any pair of neighbors u,v and any execution α. If there is a packet p in $xqueue_u[v]$ in some state s_i of α, then in constant time after s_i there is a $\text{RECEIVE}_{u,v}(p)$ event. (i.e., any packet in either the outbound queue for a link or on the link itself is delivered within constant time.)*

Proof: We know from Lemma 5.6.6 that the packet at the head of $queue_u[v]$ is placed in $Q_{u,v}$ (i.e., is placed in the channel) in t_p time and that any packet in $Q_{u,v}$ is delivered in t_l time. The lemma follows since $queue_u[v]$ has a size of at most 4, and t_p is a constant. ∎

Next, we show that within constant time the *signalbit* variable at a node becomes *false*.

Claim D.2.5 *Consider any node u,v and any execution α. If $signalbit_u$ = true in some state s_i of α, then in constant time after s_i there is a state in which $signalbit_u$ = false.*

Proof: This follows because if $signalbit_u$ = *true* in s_i, the SIGNAL_u event is enabled and $signalbit_u$ will remain *true* unless the SIGNAL_u event occurs (see code). Thus since each SIGNAL_u action is in a separate class, a SIGNAL_u action will occur in constant time after s_i, resulting in a state (see code) in which $signalbit_u$ = *false*. ∎

The next claim (which is used to show the consistency property) states the following. Suppose there is some interval in which the *mode* of a node u is *Ready* at the start and end of the interval but is not *Ready* somewhere within the interval. Consider any neighbor v of u. Then there must be some point within the interval during which an ABORT packet arrives at v; also at this point any messages in transit from u to v have been "flushed" out.

Claim D.2.6 *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$. Consider any three states s_i, s_j and s_k in α such that $i < j < k$ and $s_i.mode(u)$ = Ready, $s_j.mode(u) \neq$ Ready and $s_k.mode(u)$ = Ready. Then there is some j' such that $i < j' < k$ and $a_{j'}$ is a $\text{RECEIVE}_{u,v}(\text{ABORT},*)$ action and $s_{j'}.M_{u,v}$ does not contain any Σ-message.*

Proof: Let s_l be the first state after s_i in which $s_i.mode(u) \neq$ *Ready*. Such a state must exist by hypothesis and it must be that $l \leq j$. Thus by the code, $A3(u,v)$ is true in s_l (i.e., there is an abort packet in $xqueue_u[v]$ in s_l). Let $s_{j'}$ be the first state after s_l in which $A3(u,v)$ is *false* (i.e., the first state after s_l in which the abort packet is delivered). We know from Claim D.2.4 that such a state exists. Also, we know from $\mathcal{A}$, that in the interval $[s_l, s_{j'}]$, $mode(u) \neq$ *Ready*. Thus $j' < k$. Also $a_{j'}$ must

be a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ event and following such an event it is easy to see from the code that $mode(v) \neq Ready$ and that $buffer_v[u]$ is empty. Also we know that in the interval $[s_l, s_{j'}]$, since $mode(u) \neq Ready$ no Σ-message was added to $M_{u,v}$. Also any Σ-message in $xqueue_u[v]$ in s_l must have been removed from $xqueue_u[v]$ before $s_{j'}$ because $xqueue_u[v]$ is a FIFO queue. Thus $s_{j'}.M_{u,v}$ does not contain any Σ-message. ∎

The next claim (which is also used to show the consistency property) is a mild corollary of the previous claim. Suppose there is some interval in which $mode(u) = Ready$ at the start of the interval, and u's signal bit is false in the interval, and u does a signal at the end of the interval. Then the *mode* of u must be *Ready* at the end of the interval and must have been not *Ready* somewhere within the interval. Thus the previous claim applies, along with its consequences.

Claim D.2.7 *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Consider any state s_i such that $s_i.mode(u) = Ready$ and another state $s_{i'}$, $i' \geq i$, such that $s_{i'}.signalbit_u = false$. Suppose that after $s_{i'}$ there is a SIGNAL_u action a_k. Then there is some j where $i < j < k$ such that:*

- *a_j is a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ action.*
- *$s_j.status(v) \neq on$*
- *$s_j.M_{u,v}$ does not contain any Σ-message.*

Proof: In the state just before a_k, $signalbit_u = true$ but in s_k, $signalbit_u = false$. Let s_l be the first state before s_{k-1} in which $signalbit_u = false$. Also $l \geq i'$ because $s_{i'}.signalbit_u = false$. Thus from the code it must be that $s_l.mode(u) \neq Ready$ and $s_{l+1}.mode(u) = Ready$. The lemma follows by using Claim D.2.6 to the three states s_i, s_l and s_{l+1} and by observing that in any state that follows a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ action, $mode(v) \neq Ready$ and hence $status(v) \neq on$. ∎

Notice that the requirements of the previous claim are satisfied if $status(u) = on$ at the start of the interval and there is a signal at u at the end of the interval. We state this corollary to the previous claim as a separate claim as it is used often.

Claim D.2.8 *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Consider any state s_i such that $s_i.status(u) = on$. Suppose that after s_i there is a* SIGNAL_u *action a_k. Then there is some j where $i < j < k$ such that:*

- *a_j is a* $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ *action.*
- *$s_j.status(v) \neq on$*
- *$s_j.M_{u,v}$ does not contain any Σ-message.*

Proof: Follows from Claim D.2.7. ∎

The next three claims are all used to prove the timeliness property.

Consider any two neighboring nodes u and v. The next claim states that if the *status* of both u and v is *on* for a sufficiently large constant, then u must deliver a free event (indicating that u is willing to accept a new message to be sent to v) within this time.

Claim D.2.9 *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Consider any state s_i. Then in constant time after s_i either a* $\text{FREEM}_{u,v}$ *occurs or there is a state s_j such that either $s_j.status(u) = off$ or $s_j.status(v) = off$.*

Proof: Suppose that $status(u) = on$ and $s_j.status(v) = on$ for c time after s_i, where c is a large enough constant to make the following argument work. By $\mathcal{H}$, in state s_i either:

- $freem_u[v]$ is *true*.
- $xqueue_v[u]$ contains a $\Sigma - \text{ACK}$.
- $M_{u,v}$ contains a Σ-message.

In the first case, assuming c is large enough, $status(u) = on$ for a constant time after s_i which causes a $\text{FREEM}_{u,v}$ to occur in constant time after s_i.

In the second case, if there is a $\Sigma - \text{ACK}$ in $xqueue_v[u]$ then by Claim D.2.4 within constant time a $\text{RECEIVE}_{v,u}(\Sigma - \text{ACK})$ event occurs which causes $freem_u[v]$ to become *true*, leaving us in the first case.

For the third case, we can use an argument similar to the second case to show that in constant time after s_i, a RECEIVE$_{u,v}(m)$ event occurs which causes m to be placed in $buffer_v[u]$. Assuming again that c is large enough this implies that in constant time after s_i either a RECEIVEM$_{u,v}(m)$ event or a RECEIVE$_{u,v}$(ABORT) event occurs. Either of these events will cause a Σ – ACK to be placed in $xqueue_v[u]$, which brings us back to Case 2. ∎

Consider any two neighboring nodes u and v. The next claim states that if the *status* of both u and v is *on* for a sufficiently large constant after a message is sent from u to v, then the message will be delivered to v within this time.

Claim D.2.10 *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$.. Consider any safe* SENDM$_{u,v}(m)$ *action in α. Then in constant time after this action either a* RECEIVEM$_{u,v}(m)$ *occurs or there is a state s_j such that either $s_j.status(u) =$ off or $s_j.status(v) =$ off.*

Proof: Similar to proof of Claim D.2.9. Let us denote the safe SENDM$_{u,v}(m)$ event by a_j. Suppose that $status(u) = on$ and $s_j.status(v) = on$ for c time after a_j, where c is a large enough constant to make the following argument work. Then since a_j is a safe send it is easy to see from the code that $freem_u[v]$ is *true* in the state before a_j. Also by our assumption, $status(u) = on$ in the state after a_j. So m is placed in $queue_u[v]$ in the state after a_j. Thus by Claim D.2.4, in constant time after a_j, m is placed in $buffer_v[u]$ and if $status(v) = on$ for constant time after this, a RECEIVEM$_{u,v}(m)$ event occurs. ∎

D.2.2 Every Behavior of $\mathcal{R}|L$ is timely

We prove that every behavior β of $\mathcal{R}|L$ is timely by showing each of the four properties in Definition 7.3.3. Corresponding to the four properties, we have four lemmas.

We will leave the first property of a timely behavior (i.e., that all messages received after $O(n)$ time are normal) to the end of this section. We start by showing the second property.

Lemma D.2.11 Periodic Free Events: *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$ and any state s_j in α. Then either a* FREEM$_{u,v}(m)$ *occurs*

in constant time after s_j or a SIGNAL_u *action occurs in $O(n)$ time after s_j or or a* SIGNAL_v *action occurs in $O(n)$ time after s_j.*

Proof: We know from Claim D.2.9 that for some constant c, in c time after s_j a $\text{FREEM}_{u,v}(m)$ occurs or there is a state s_k such that either $s_k.status(u) = off$ or $s_k.status(v) = off$. In the first case, we are done. In the second case, we know from the Signal Lemma (Lemma D.2.2), that within $O(n)$ time after s_k either a SIGNAL_u or a SIGNAL_v event occurs, ∎

Next, we prove that $\mathcal{R}|L$ satisfies the third property of a timely behavior.

Lemma D.2.12 Timely Message Delivery: *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$. Consider any a_j that is a safe* $\text{SENDM}_{u,v}(m)$ *action in α. Then either a* $\text{RECEIVEM}_{u,v}(m)$ *occurs in constant time after a_j or a* SIGNAL_u *action occurs in $O(n)$ time after a_j or or a* SIGNAL_v *action occurs in $O(n)$ time after a_j.*

Proof: We know from Claim D.2.10 that for some constant c, in c time after a_j a $\text{RECEIVEM}_{u,v}(m)$ occurs or there is a state s_j such that either $s_j.status(u) = off$ or $s_j.status(v) = off$. In the first case, we are done. In the second case, we know from the Signal Lemma (Lemma D.2.2) that within $O(n)$ time after s_j either a SIGNAL_u or a SIGNAL_v event occurs, ∎

Next, we prove that $\mathcal{R}|L$ satisfies the fourth property of a timely behavior.

Lemma D.2.13 Signals at a Node induce Signals at Neighbors*: Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$. There is some constant c such that for every* SIGNAL_u *event a_j that occurs at time greater than $\alpha.start + c \cdot n$ there is a* SIGNAL_v *event that occurs in linear time before or after a_j.*

Proof: First, within linear time of the start of β, there must be some state s_h in which $mode(u) = Ready$ (by the Termination Lemma). In constant time after s_h, there must be some state s_i in which $signalbit_u = false$ by Claim D.2.5. Consider any SIGNAL_u action a_j that occurs after state s_i. In the state before a_j, $signalbit_u = true$ but in state s_i, $signalbit_u = false$. Consider the first state s_k before s_j in which $signalbit_u = false$. By Claim D.2.5, s_j occurs in constant time after s_k. Also since $s_{k+1}.signalbit_u = true$,

the code tells us that that $s_k.mode(u) \neq ready$. But we know that $s_h.mode(u) = Ready$. Consider the first state s_l before s_k in which $mode(u) = Ready$. By the Termination Lemma, s_l occurs in linear time before s_k.

Thus we have identified two states s_l and s_k, with $k \geq l$ such that both states occur before the SIGNAL_u event a_j and such that $s_l mode(u) = Ready$ and $s_k.signalbit_u = false$. By applying Claim D.2.7 to s_l, s_k and a_j we know that there is some state s_m in the interval $[s_l, s_j]$ in which $status(v) \neq on$. Thus by the Signal Lemma, a SIGNAL_v event occurs within linear time after s_m. But since s_l occurs within linear time before s_j, the SIGNAL_v event occurs in linear time before or after a_j. ∎

Finally, we prove that $\mathcal{R}|L$ satisfies the first property of a timely behavior. This requires more work and so we start with two claims. The claims are quite intuitive.

The first claim states that in any suffix of an execution, all except possibly the first packet received on a link are normal packets that have been sent in this execution.

Claim D.2.14 *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Let s_i be the first state in α such that there is no Σ-message in $s_i.M_{u,v}$. Consider any a_k that is a $\text{RECEIVE}_{u,v}(m)$ event that occurs after s_i in α. Then:*

- *There is a $\text{SEND}_{u,v}(m)$ action a_j before a_k and such that there are no $\text{RECEIVE}_{u,v}(*)$ events in between a_j and a_k. We will call the earliest such a_j the send corresponding to a_k in α.*
- *In state s_j (i.e., the state immediately after a_j in α), $status(u) = on$ and in state s_k, $status(v) = on$*
- *In all states in the interval $[s_j, s_{k-1}]$, m is in $M_{u,v}$.*

Proof: It is clear from the code that in s_{k-1}, m must be in $buffer_u[v]$ and hence m is in $M_{u,v}$. But since in s_i, $M_{u,v}$ is empty, there must be a $\text{SEND}_{u,v}(m)$ action between s_i and s_{k-1} which added m to $M_{u,v}$. Let a_j be the first such action that occurs before a_k. In the state immediately after a_j by the code $status(u) = on$. Clearly, there cannot be a $\text{RECEIVE}_{u,v}(*)$ action between a_j and a_k because (from $\mathcal{H}$), $M_{u,v}$ contains at one most one message in any state. But a $\text{RECEIVE}_{u,v}(m)$ action is the only action that can remove m from $M_{u,v}$ and hence m is in $M_{u,v}$ in the interval $[s_j, s_{k-1}]$. Also from the code, in the state immediately after a $\text{RECEIVE}_{u,v}(m)$ action, $status(v) = on$. ∎

Claim D.2.15 *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$ and any state s_i in α. Then there is some state s_m that occurs in $O(n)$ time after s_i such that there is no Σ-message in $s_m.M_{u,v}$.*

Proof: From Lemma D.2.1, there is a state s_j that occurs in $O(n)$ time after s_i such that $s_j.mode(v) = Ready$. From Claim D.2.5, in constant time after s_j there is a state $s_{j'}$ in which $signalbit_v = false$.

From Lemma D.2.1, there is a state s_k that occurs in $O(n)$ time after $s_{j'}$ such that $s_k.mode(u) = Ready$. From Claim D.2.5, in constant time after s_k there is a state $s_{k'}$ in which $signalbit_u = false$.

From Lemma D.2.11, some event a_l occurs within $O(n)$ time after $s_{k'}$, where a_l is either a $\text{FREEM}_{u,v}$ event or SIGNAL_u or a SIGNAL_v event. In the first case, we are done by predicate $\mathcal{H}$ which shows that $\text{FREEM}_{u,v}$ cannot occur unless $M_{u,v}$ is empty. In the second case a_l is a SIGNAL_u event, then by Claim D.2.7, there is some state $s_{l'}$ in the interval $[s_k, s_l]$ such that there is no Σ-message in $s_k.M_{u,v}$. If a_l is a SIGNAL_v event, then by Claim D.2.7 there is a state say $s_{l'}$ in the interval $[s_j, s_l]$ such that $s_{l'}.status(u) = off$. Thus by the Signal Lemma (Lemma D.2.2), a SIGNAL_u event occurs within $O(n)$ time after $s_{l'}$, which brings us back to the second case. ∎

We can now show the first part of the consistency property (see Definition 7.3.5), that every message received after $O(n)$ time is normal; it is almost immediate from the last two claims.

Lemma D.2.16 Normal Receipt of Messages: *Consider any any execution α of $\mathcal{R}|L$ and any suffix γ of α with first state s_0. There is some constant c such that every every receive event that occurs at time greater than $s_0.time + c \cdot n$ in α 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.*

Proof: Consider any pair of neighbors u, v. By Claim D.2.15, there is some s_j that occurs in $O(n)$ time after s_0 and such that there is no Σ-message in $s_j.M_{u,v}$. Let us call j the quiescent index for link (u, v). Further, let k be the largest quiescent index over all possible links (u, v). Clearly s_k occurs in $O(n)$ time after s_0. Also by Claim D.2.14, all receive events that occur after s_k in γ are normal.

Also let a_j be any normal receive event and a_i be the send corresponding to a_j. By Claim D.2.14, in all states in the interval $[s_i, s_{j-1}]$, m is in $M_{u,v}$. But by Claim D.2.15, there is some state s_k that occurs in $O(n)$ time after s_i such that there is no Σ-message in $s_k.M_{u,v}$. Thus a_j occurs within $O(n)$ time after a_i. ∎

And now we come to the main result of this subsection:

Lemma D.2.17 *Every behavior of $\mathcal{R}|L$ is timely.*

Proof: Immediate from Definition 7.3.3 and Lemmas D.2.16, D.2.13, D.2.11, and D.2.12. ∎

D.2.3 Every Behavior of $\mathcal{R}|L$ satisfies the consistency property

We prove that every behavior β of $\mathcal{R}|L$ satisfies the consistency property by showing each of the five properties in Definition 7.3.5. We have the following preliminary claim that is essential for proving the consistency property.

The claim states the following. Consider some interval and some node u and suppose that the *status* of u is *on* at the start of the interval and the interval ends with u receiving a normal message m from v. Suppose also that there is a SIGNAL_u event in the interval. Then from Claim D.2.8 we know that there is some point within the interval during which an ABORT packet arrives at v and such that any messages in transit from u to v have been "flushed" out. However. this claim goes further and states that the point at which the ABORT packet arrives at v occurs *before* v sends message m.

Claim D.2.18 *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Consider any i, j, k such that $i < j < k$. Suppose $s_i.status(u) = on$, a_j is a SIGNAL_u event, and a_k is a normal receive event at u from v. Let $a_{k'}$ be the send corresponding to a_k. Then there is some $i < j' < k'$ such that $status(v) \neq on$ and $s_{j'}.M_{u,v}$ is empty.*

Proof: By Claim D.2.8 applied to s_i and a_j we know there is some j' where $i < j' < j$ such that:

- $a_{j'}$ is a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ action.
- $s_{j'}.mode(v) \neq Ready$
- $s_{j'}.M_{u,v}$ does not contain any Σ-message.

As long as we can prove that $j' < k'$ we are done. Suppose not. By $\mathcal{A}$ we know that $s_{j'}.ack_u[v] = true$. But since a_k is a receive action, $s_k.status(u) = on$, and so we know that $s_k.ack_u[v] = false$. Let s_l be the first state after $s_{j'}$ such that $s_l.ack_u[v] = false$. Thus $j' < l < k$. But we know from the code that $ack_u[v]$ is only set to *false* after a $\text{RECEIVE}_{v,u}(\text{ACK})$ event and so a_l must be a $\text{RECEIVE}_{v,u}(\text{ACK})$ event. But by $\mathcal{B}$ we know that there is no ACK packet in $M_{v,u}$ in state $s_{j'}$. Thus there must be some n, $j' \leq n < k$ such that $a_n = \text{SEND}_{v,u}(\text{ACK})$. Intuitively, what we have shown is that in the interval $[s_{j'}, s_k]$ an ack packet must have been sent by v and received by u.

Now we can obtain the required contradiction. We will only sketch the rest of the argument informally. Suppose for contradiction, that $j' \geq k'$. Then it must be that the ack sent (in action a_n) was sent *after* the user message sent (in action $a_{k'}$). But then (essentially because the channels and queues are FIFO), the corresponding user message receipt (i,e., action a_k) must have occurred before the corresponding ack packet receipt (call this action a_l). Thus $k < l$. Also, since $j' < k$, we know that k lies in the interval $[s_{j'}, s_{l-1}]$. But we know from $\mathcal{A}$ that in the interval $[s_{j'}, s_{l-1}]$, $mode(u) \neq Ready$ since in this interval u is still waiting for an ack from v. But this contradicts the fact that in a state s_k immediately following a receive event such as a_k, $mode(u)$ must be *Ready*. ∎

Next, we show a lemma (see Figure D.1) which states (in essence) that messages sent in a signal interval at v can be received in at most one signal interval at u; conversely messages received in a signal interval at u could have been sent in at most one signal interval at v. This will help establish that each signal interval at u can have at most one mated signal interval at v.

Lemma D.2.19 Send Consistency: *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$. Let a_j and a_k be any two normal receive events at u from v in α. Let a_l and a_m be the send events corresponding to a_j and a_k respectively. Then there is a SIGNAL_v event between a_l and a_m iff there is a SIGNAL_u event between a_j and a_k.*

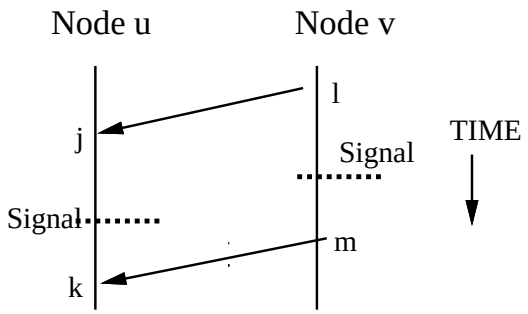

Figure D.1: Send consistency: there is a signal between the two receives at u iff there is a signal between the two corresponding sends at v.

Proof: Assume without loss of generality that $j < k$. Suppose there is a SIGNAL$_v$ event, say $a_{m'}$ between a_l and a_m. By Claim D.2.14, $s_l.status(v) = on$. Then by Claim D.2.8 there must be some p, $l < p < m'$, such that $s_p.status(u) = off$ and $M_{v,u}$ is empty. But by Claim D.2.14, $M_{v,u}$ is non-empty in the interval $[s_l, s_j]$. Thus since $p > l$, it must be that $p > j$. Also $p < m'$ and $m' < m$ and by Claim D.2.14, $m < k$. So $p < k$. Thus there is a state s_p that occurs in the interval $[s_j, s_k]$ in which $status(u) = off$. Also we know by Claim D.2.14 that $s_k.status(u) = on$. Thus by Claim D.2.3, a SIGNAL$_u$ event must occur in the interval $[s_j, s_k]$.

The reverse argument is slightly different. Suppose there is a SIGNAL$_u$ event, say $a_{k'}$ between a_j and a_k. By the code, $s_j.status(v) = on$. Thus by Claim D.2.18, applied to s_j, $a_{k'}$ and a_k we know that there is some j' such that $j < j' < m$ and $s_{j'}.status(v) \neq on$. But since $l < j$, $s_{j'}$ occurs in the interval $[s_l, s_m]$ and $status(v) = off$. Also we know by Claim D.2.14 that $s_m.status(v) = on$. Thus by Claim D.2.3, a SIGNAL$_u$ event must occur in the interval $[s_l, s_m]$. ∎

Next, we show a second lemma (see Figure D.2) which states (in essence) that a signal interval at u cannot send messages to and receive messages from different signal intervals at v. This will help show that the mating relation is symmetric

Lemma D.2.20 Send-Receive Consistency: *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Let a_j be a normal receive event at u from v and let a_m be a normal receive event at v from u. Let a_l and a_k be the send events corresponding to a_j and a_m respectively. Then there is a* SIGNAL$_v$ *event between a_l and a_m iff there is a* SIGNAL$_u$ *event between a_j and a_k.*

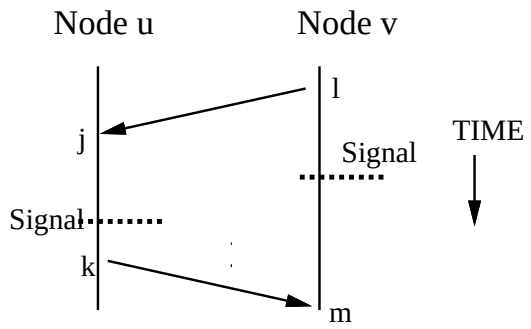

Figure D.2: Send-Receive consistency: there is a signal between the receive and the send at u iff there is a signal between the corresponding send and receive at v.

Proof: Assume that $k > j$ as shown in the Figure D.2. The other cases are similar.

Suppose there is a SIGNAL$_u$ event, say $a_{k'}$ between a_j and a_k. By Claim D.2.14, $s_j.status(u) = on$. Then by Claim D.2.8 there must be some p, $j < p < k'$, such that $s_p.status(v) = off$. Thus since $p > j$, it must be that $p > l$. Also $p < k$ and $k < m$. So $p < m$. Thus there is a state s_p that occurs in the interval $[s_l, s_m]$ in which $status(v) = off$. Also we know by Claim D.2.14 that $s_m.status(v) = on$. Thus by Claim D.2.3, a SIGNAL$_v$ event must occur in the interval $[s_l, s_m]$.

The reverse argument is slightly different. Suppose there is a SIGNAL$_v$ event, say $a_{m'}$ between a_l and a_m. By the code, $s_l.status(v) = on$. Thus by Claim D.2.18, applied to s_l, $a_{m'}$ and a_m we know that there is some p such that $p < k$ and $s_p.status(u) \neq on$ and such that $s_p.M_{v,u}$ does not contain a Σ-message. We claim that $j < p$. If not not s_p must lie in the interval $[s_l, s_j]$ and in this interval we know that there is always a Σ-message in $M_{v,u}$ by Claim D.2.14. But this contradicts the fact that $s_p.M_{v,u}$ does not contain a Σ-message.

Thus s_p occurs in the interval $[s_j, s_k]$ and $status(u) \neq on$. Also we know by Claim D.2.14 that $s_k.status(v) = on$ Thus by Claim D.2.3, a SIGNAL$_u$ event must occur in the interval $[s_j, s_k]$. ∎

We now show the third part of the consistency property (see Definition 7.3.5).

Lemma D.2.21 Successful Sending of Messages: *Consider any pair of neighbors u, v and any execution α of $\mathcal{R}|L$. Between any safe* SENDM$_{v,u}(m)$ *event and a later* FREEM$_{v,u}$ *event, there is either a* RECEIVEM$_{v,u}(m)$ *event or a* SIGNAL$_v$ *event.*

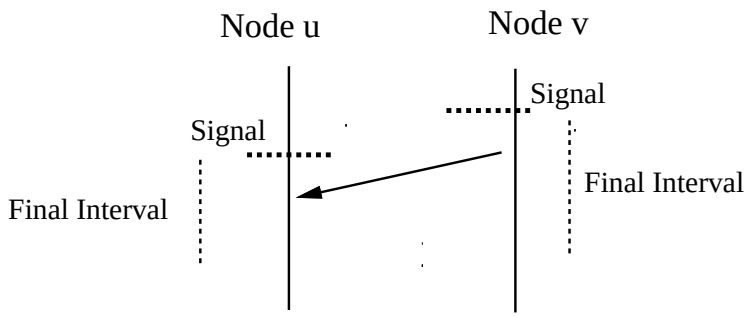

Figure D.3: Mating of Final Signal Intervals: Messages sent in a final interval can only be received in another final interval.

Proof: Let us denote the $\text{SENDM}_{v,u}(m)$ event by a_i and the $\text{FREEM}_{v,u}$ event by a_k. Clearly $l < k$.

Thus, by the code $status(v) = on$ in s_k. Also by the code, $freem_v[u] = true$ in s_k and hence by $\mathcal{H}$, $M_{v,u}$ does not contain any Σ-message in s_k.

Next, consider a_i. Since a_i is safe, by definition there must an action $a_h = \text{FREEM}_{v,u}(m)$ such that $h < i$ and such that there is no other $\text{SENDM}_{v,u}(*)$ action between a_h and a_i. Thus, by the code $freem_v[u] = true$ in s_h and s_{i-1}. Thus, by the code, we see that either $s_i.status(v) = off$ or m belongs to $M_{v,u}$ in s_i (i.e., the message m is placed on the queue at v to send to u). But in the first case, we are done by Claim D.2.3, which tells us there must be a SIGNAL_v event between s_i and s_k.

So consider the second case where m belongs to $M_{v,u}$ in s_i and $s_i.status(v) = on$. But we know that in the later state s_k, m does not belong to $M_{v,u}$. Now, from the code, the only two actions that could remove m from $M_{v,u}$ in the interval $[s_i, s_k]$ are a $\text{RECEIVEM}_{v,u}(m)$ event or a $\text{RECEIVE}_{v,u}(\text{ABORT}, *)$ event. In the former case, we are done; so consider the latter case. Now, because $s_i.mode(v) = on$, we know from $\mathcal{A}$ that there is no $(\text{ABORT}, *)$ packet in $s_i.xqueue_v[u]$. Thus there must have been an action $a_j = \text{SEND}_{v,u}(\text{ABORT}, *)$ in the interval $[s_i, s_k]$. From the code, $s_j.status(v) = off$. The lemma now follows from Claim D.2.3, which tells us there must be a SIGNAL_v event between s_j and s_k. ∎

We now show the fourth part of the consistency property (see Definition 7.3.5). The lemma on which it is based is sketched in Figure D.3.

Lemma D.2.22 Mating of Final Signal Intervals: *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$. Let a_j be a normal receive event at u from v in α and a_l be the send corresponding to a_j. Then there is no* SIGNAL_u *event after a_j iff there there is no* SIGNAL_v *event after a_l.*

Proof: Suppose there is a SIGNAL_v event a_n after a_l. Then by Claim D.2.8, there is a state s_m between a_l and a_n such that $s_m.mode(u) \neq Ready$ and $s_m.M_{v,u}$ does not contain any Σ messages. It follows from Claim D.2.14 that s_m must occur after a_j. Thus there is a state after a_j in which $mode(u) \neq Ready$. Thus by the Signal Lemma (Lemma D.2.2), a SIGNAL_u event will occur after a_j.

The reverse argument is similar but slightly simpler. ∎

Lemma D.2.23 Mating Relation Preserves Temporal Ordering: *Consider any pair of neighbors u,v and any execution α of $\mathcal{R}|L$. 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.*

Proof: Omitted. Follows, in essence, from the FIFO properties of the underlying UDLs and the fact that ABORT packets sent between signal intervals flush the links and buffers of previously sent messages. ∎

And now we come to the main result of this subsection:

Lemma D.2.24 *Every behavior of $\mathcal{R}|L$ satisfies the consistency property.*

Proof: We define a signal interval at a node u in an execution α by analogy with the definitions for behaviors. We define two signal intervals I_u at u and I_v at v to be mates if any normal message received in I_u was sent in I_v or vice versa. Lemma D.2.19 and Lemma D.2.20 show that the mating relation is well-defined and that any signal interval at u can have at most one mate and that the relation is symmetric. The last three conditions in Definition 7.3.5 follow from Lemma D.2.21, Lemma D.2.22 and Lemma D.2.23. ∎

D.2.4 Every Behavior of $\mathcal{R}|L$ is causal

We can now prove that the Reset protocol is causal. We first prove the first property in Definition 7.3.6.

Theorem D.2.25 *Consider any execution $\alpha = s_0, a_1, s_1, \ldots$ of $\mathcal{R}|L$. There is some constant c such that every signal event a_k that occurs at time greater than $s_0.time + cn$ is preceded by a request event a_j such that $a_k.time - a_j.time \leq cn$.*

Proof: A formal proof can be patterned after the intuitive argument given in Section 7.7. ∎

Next, we prove the second property in Definition 7.3.6.

Lemma D.2.26 *Consider an execution $\alpha = s_0, a_1, s_1, \ldots$ of $\mathcal{R}|L$. There is some constant c such that a* SIGNAL$_u$ *event occurs within cn time of any* REQUEST$_u$ *event.*

Proof: We know from the code that in the state immediately following a REQUEST$_u$ event, $status(u) = off$. The lemma follows immediately from the Signal Lemma (Lemma D.2.2). ∎

And now we come to the main result of this subsection:

Lemma D.2.27 *Every behavior of $\mathcal{R}|L$ is causal.*

Proof: Immediate from Definition 7.3.6 and Lemmas D.2.25 and D.2.26. ∎

D.2.5 The Main Theorem

Finally, we can state our main theorem of this section, which follows from the main lemmas of the last three subsections:

Theorem D.2.28 *Every behavior of $\mathcal{R}|L$ is in RP.*

Proof: Immediate from Definition 7.3.7 and Lemmas D.2.17, D.2.24, and D.2.27. ∎

Appendix E

Dijkstra's Token Protocol as an Example of Counter Flushing

In Chapter 10, we described a paradigm called counter flushing. We now show that Dijkstra's first example protocol in [Dij74] can be simply understood using this paradigm.

Dijkstra's first example is modelled by the automaton $D2$ shown in Figure E.1. As in the previous example, the nodes (once again numbered from 0 to $n-1$) are arranged such that node 1 has node 0 and node 2 as its neighbors and so on. However, in this case we also assume that Process 0 and $n-1$ are neighbors. In other words, by making 0 and $n-1$ adjacent we have made the line into a ring. For process i, let us call Process $i-1$ (we assume that all arithmetic on indices and counters is mod n) the anticlockwise neighbor of i and $i+1$ the clockwise neighbor of i.

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

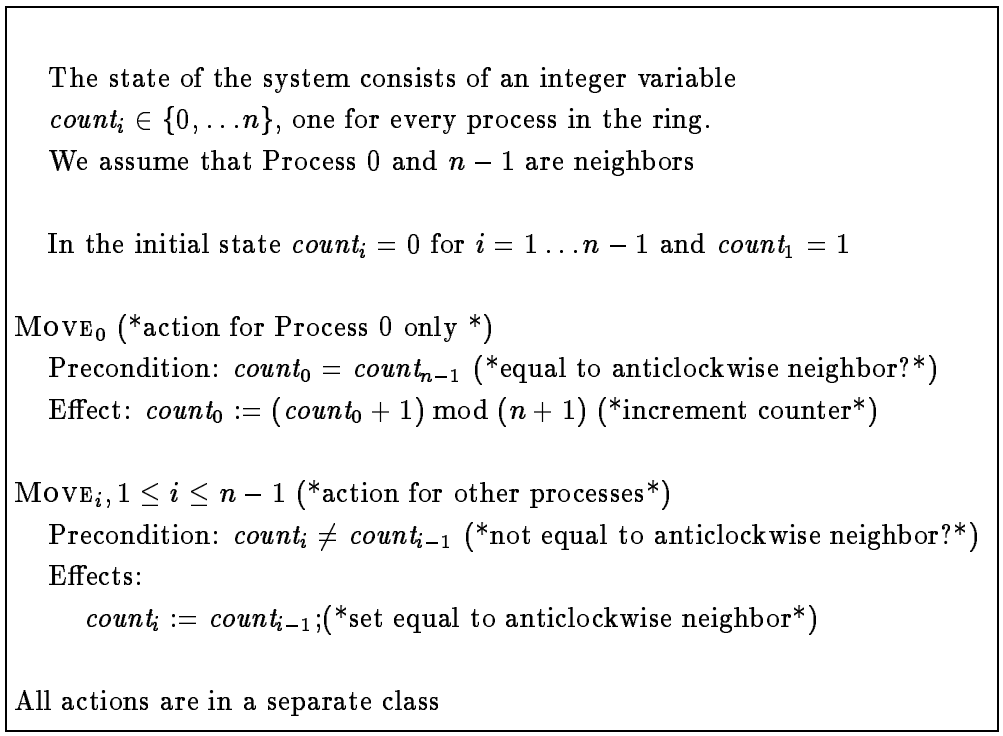

Figure E.1: Automaton $D1$: a version of Dijkstra's first example with initial states. The protocol does token passing on a ring using nodes with n states.

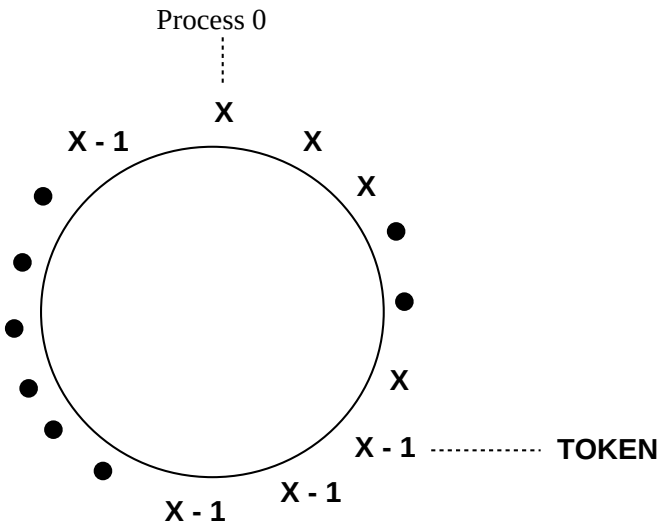

Figure E.2: In the good states for Dijktra's first example, the ring can be partitioned into 2 bands with the token at the boundary.

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

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

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

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

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

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

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

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

The net effect is that any execution of $D1$ eventually reaches a good state in which it remains. The reader should compare our proof of this protocol with the general description of counter flushing found in Chapter 10.

Bibliography

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

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

[AE86] Baruch Awerbuch and Shimon Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16(4):381–396, Winter 1986.

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

[AG91] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 1991.

[AG92] Anish Arora and Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. Unpublished manuscript, February 1992.

[AGLP89] Baruch Awerbuch, Andrew Goldberg, Michael Luby, and Serge Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, May 1989.

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

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

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

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

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

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 206–220, October 1988.

[AS90] Baruch Awerbuch and Mike Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

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

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

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

[BGW87] G.M. Brown, M.G. Gouda, and C.L. Wu. A self-stabilizing token system. In *Proceedings of 20th Hawai International Conference on System Sciences*, pages 218–223, 1987.

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

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

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

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

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

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

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

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

[DIM91a] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message driven protocols. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, Montreal, Canada, August 1991.

[DIM91b] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election. Unpublished manuscript, 1991.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Info. Process. Letters*, 11(1):1–4, August 1980.

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

[Gal76] Robert G. Gallager. A shortest path routing algorithm with automatic resynch. Technical report, MIT, Lab. for Information and Decision Systems, March 1976.

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

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

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

[Gol85] Andrew V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, MIT, Lab. for Computer Science, November 1985.

[GPS87] A. V. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry breaking in sparse graphs. In *Proc. 19th ACM Symp. on Theory of Computing*. ACM SIGACT, ACM, May 1987.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *J. of the ACM*, 35(4):921–940, October 1988. Also appeared in 18th stoc (1986).

[IJ90] Amos Israel and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

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

[LA90] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5(2):190–222, April 1983.

[Lam84] L. Lamport. Solved problems, unsolved problems, and non-problems in concurrancy. In *Proc. of the 3rd ACM Symp. on Principles of Distributed Computing, Vancouver, BC, Canada*, pages 1–11, Aug 1984. Invited address at the 2nd ACM Symp. on Principles of Distributed Computing, Montreal, Canada, August 1983.

[Lau90] Tony Lauck. personal communication. unpublished, 1990.

[Lin87] Nathan Linial. Locality as an obstacle to distributed computing. In 27^{th} *Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[LS91] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 320–330. ACM/SIAM, January 1991.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

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

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

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

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

[MPV90] Yishay Mansour, Boaz Patt, and George Varghese. Self-stabilizing data structures. unpublished manuscript, December 1990.

[MRR80] John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Commun.*, 28(5):711–719, May 1980.

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

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

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

[Per88] Radia Perlman. *Network Layer Protocols With Byzantine Robustness*. PhD thesis, MIT, Laboratory for Computer Science, August 1988.

[RF89] Balasubramanian Rajagopalan and Michael Faiman. A new responsive distributed shortest path routing algorihtm. In *Proceedings of the Annual ACM SIGCOMM*

Symposium on Communication Architectures and Protocols, Austin, Texas, pages 237–246. ACM SIGCOMM, ACM, September 1989.

[RL81] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th POPL*, pages 133–138, 1981.

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

[SG89] John M. Spinelli and Robert G. Gallager. Broadcasting topology information in computer networks. *IEEE Trans. on Commun.*, May 1989.

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

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

Index

Index

actions, 42
    disabled, 43
    enabled, 43
    external, 42
    input, 42
    internal, 42
    locally controlled, 42
    output, 42
augmented topology graph, 240
automaton, 42
    behavior, 44
    classes, 42
      upper bounds, 42
    execution, 43
    untimed behavior, 44
    untimed execution, 44
automaton for augmented graph, 240
automaton for graph
    network model, 105

behavior sequence, 44

causality, 167
checker
    for synchronous protocols, 243
CIOA , 62
C, all edges clean, 127
clean edge, 125
closed local predicate, 100
closed predicate, 78
compatible automata, 45
composition, of automata, 45
compositional model, 46
conjunction, 101
consistency, 165
countset, 124
counter flushing, 115, 268

drop-free, 124

event, 43

ghost root, 228
Global Correction Theorem, 208

height, of partial order, 104
hiding, 47

I/O relation, 241
initialized signal interval, 214
interactive protocols, 239
interval of a behavior, 47
interval of an execution, 47

leader edge, 92
    corresponding to an edge, 290
link delay, 93
link predicate set
    network model, 101
    shared memory model, 78

link subsystem
    network model, 100
    shared memory model, 78
Local Correction Theorem, 106
local predicate
    separable, 231
    shared memory model, 78
local reset function
    network model, 104
locally checkable
    network model, 101
    shared memory model, 78
locally correctable
    network model, 104
locally extensible
    network model, 147
    shared memory model, 81

mating relation, 165
    transitivity, 168
message-renaming, 207
mode
    of a node in reset protocol, 173

network automaton
    network model, 97
    shared memory model, 77
node automaton, 96
node delay, 96
non-interactive protocol, 241
non-interactive protocols, 239
normal receive, 163

one-way checkable, 231
one-way correctable, 268
one-way predicate, 231
output function, 241

$P_{control}$, 93
P_{data}, 93
phase, 121
    mode, 288
    well-behaved, 290
predicate, 47
    holds, 47
    is true, 47
    remains true, 47
$Proj$, 120

quiescent state, 213
Q, all edges quiet, 129
quiet edge, 128
$\mathcal{Q}$, quiet edge predicate set, 129

refinement mapping
    theorem, 57
regular message receipt, 214
reqcount, 124
reset transition, 286
reset trigger, 213
respcount, 124

safe, 162
satisfies, 100
self-stabilizes, 51
send
    corresponding to normal receive event, 163
signal interval, 165
solve, 52
space complexity, 242
stabilization time, 54

with respect to I/O relation, 242
stabilized using predicate set, 60
stabilizes
to behaviors in set, 53
to behaviors of another IOA, 54
to executions in set, 50
to executions of another IOA, 51
to I/O relation, 242
stable tree behaviors
used to define spanning tree protocol correctness, 222
status
of a node in reset protocol, 177
suffix-closed, 62
synchronous protocol, 242
space complexity, 243
time complexity, 243

t-suffix
of behavior, 53
of execution, 50
t_l, default link delay, 97
t_p, time for a phase, 122
t_n, default node delay, 97
time sequence, 43
start, 43
t_q, time for a link to become quiet, 131
timed element, 43
timed action, 43
timed state, 43
timeliness, 164
timer flushing, 221, 234, 269
topology graph, 92
tree automaton
network model, 137
shared memory model, 77
Tree Correction Theorem, 148
tree graph, 137

$U(A)$, 51
UIOA , 61
unexpected packet transition, 148
Unit Storage Data Link (UDL), 93

weakly checkable, 147