Self-stabilization by Counter Flushing

George Varghese
Dept. of Computer Science
Washington University in St. Louis
St. Louis, MO 63130

August 19, 1998

Abstract

A useful way to design simple and robust protocols is to make them self-stabilizing. A protocol is said to be self-stabilizing if it begins to exhibit correct behavior even after starting in an arbitrary state. We describe a simple technique for self-stabilization called *counter flushing* which is applicable to a number of distributed algorithms. We show how our technique helps to understand and improve some previous distributed algorithms. We also apply it to create *new* self-stabilizing protocols for propagation of information with feedback and resets. The resulting protocols are simple, require few changes from the non-stabilizing equivalents, and have fast stabilization times.

Keywords. distributed algorithms, self-stabilization

AMC subject classifications. 68Q22, 68Q60, 68Q25

1 Introduction

As the world moves from an industrial economy to an information based economy, we are already dependent on networks and will become even more so. Most users of data networks, however, agree that current data networks are far more unreliable than say the telephone network. This must change.

The current approach to network fault-tolerance is what we call the *piecemeal* approach. The protocol designer enumerates the faults that the network will deal with – e.g., node and link crashes, bit errors on links — and adds recovery mechanisms for each such fault. This often adds complexity as the mechanisms are not orthogonal and often have subtle interactions.

Also, there are a number of more obscure errors (e.g., memory corruption) that occur in real networks and are hard to anticipate and enumerate. Even if such faults occur rarely (say once a month), it makes economic sense to have networks automatically recover from such faults.

These issues are illustrated by the crash of the original ARPANET protocol ([Ros81],[Per83]). The protocol was designed never to enter a state that contained three conflicting updates. Unfortunately, a malfunctioning node injected three such updates into the network and crashed. After this the network cycled continuously between the three updates. It took days of detective work [Ros81] and much coast-to-coast coordination before the problem was diagnosed.

Self-stabilization: Ideally networks should recover by themselves, even from obscure faults. As networks grow faster and are more commonly used, the likelihood of such obscure faults occurring will increase. This paper describes a paradigm for designing what are known as *self-stabilizing* network protocols. We do so to make network protocols *simpler* (i.e., a uniform mechanism instead of a slew of mechanisms to deal with a catalog of anticipated faults) and *more robust* (e.g., can recover from transient faults such as memory corruption as well as common faults such as link and node crashes).

We model a computer network as a set of nodes that are interconnected by communication channels. A network protocol consists of a program for each network node. Each program consists of code and inputs as well as local state. The global state of the network consists of the local state of each node as well as the messages on network links. A network protocol is *self-stabilizing* if when started from an arbitrary state it exhibits "correct" behavior after finite time. This definition allows arbitrary corruption of messages and node state variables in the initial state.

Note that we allow network *state* to be corrupted but not the *code* executing the protocol. This is reasonable because program code can be protected against arbitrary corruption of memory by redundancy since code is rarely modified. For example, the code can be checksummed. However, it is not clear how one can detect corruption of network state (that is frequently being modified) by using redundancy techniques.

The definition seems to imply that faults can occur only once (i.e., when the network "starts"). However, the real assumption is: *the average period between faults covered by self-stabilization is larger than the time the protocol takes to stabilize.*

The distributed algorithm literature also describes *Byzantine* fault models ([LSP82]) in which arbitrary faults can continuously occur. This may appear to be a more general model than self-stabilization. However, in Byzantine models, only a fraction of nodes are allowed to exhibit arbitrary behavior. In the self-stabilization model, *all* nodes are permitted to start with arbitrary initial states. Thus, the two models are orthogonal. In a practical setting the crucial difference is that the cost of stabilization is quite cheap while Byzantine solutions are expensive. For example, the self-stabilizing routing protocol in [Per83] is much cheaper than the Byzantine routing protocol of [Per88].[1]

General Techniques for Self-stabilization: Self-stabilizing protocols were introduced by Dijkstra [Dij74b], and have been studied by various researchers. [Sch93] contains a recent review. While a large number of *ad hoc* self-stabilizing protocols have been introduced, there have been few general techniques for self-stabilization. Katz and Perry [KP93] showed how to compile an arbitrary asynchronous protocol into a stabilizing equivalent. Their general transformation is expensive; hence more efficient (and possibly less general) techniques are needed. Techniques that transform any *locally checkable* protocol into a stabilizing equivalent are given in [AGV94, Var93]. However, local checking only applies to a subset of protocols that have a special property called *local checkability.*

Our paper describes a new general technique, called *counter flushing* that is applicable to protocols that are not locally checkable. The setting is that of a leader who wishes to periodically deliver a message to every network node (and sometimes to every link) in the network. By attaching a counter to the state of every node and to every message, and by using a few simple checks, we ensure that the protocol will begin to work correctly regardless of the initial messages and node states. The paradigm is also simple (in terms of the lines of code added and the resulting proofs), and provides fast stabilization times (equal to a few round trip delays).

Counter flushing can be applied to a variety of useful distributed applications. In particular, counter flushing applies to several *total algorithms.* Total algorithms [Tel89] involve the cooperation of all network nodes. We apply counter flushing to token passing [Dij74a], propagation of information with feedback [Seg83], and network resets [AG94].

The rest of the paper is organized as follows. Section 2 describes our model. Section 3 describes how counter flushing works on ring topologies and shows how this algorithm can be used for token passing, request-response, and Data Link protocols. Section 4 describes how counter flushing works on trees by describing a stabilizing broadcast protocol called Propagation of Information with Feedback. In Section 5, we extend counter flushing to a general graph. We illustrate this by designing a stabilizing network reset protocol. In Section 6, we present a uniform proof of stabilization and correctness of our three main protocols. The uniformity of proof emphasizes the unity behind the diversity of applications. We conclude in Section 7.

[1] The protocol in [Per83] is the basis for the OSI Routing Protocol and the OSPF Routing protocol, both of which are widely deployed; we know of no Byzantine routing protocol that is used in a real network.

2 Model

We restrict ourselves to message passing protocols for networks. The network topology is modeled by a directed graph $G = (V, E)$. Let $n = |V|$ denote the number of network nodes and D the network diameter. Except for the case when we consider a ring topology (Section 3), we assume that the graph is symmetric – i.e., if $(i, j) \in E$ then $(j, i) \in E$. We assume there is a distinguished leader node $0 \in V$ that we often refer to as the *root*. Note that there are many stabilizing protocols to construct a leader; e.g., [AKM$^+$93, Dol94, AK93] all calculate a leader in $O(D)$ time. Finally, we model the nodes and links of the network using Input/Output Automata (IOA) [LT89].

An IOA is a state machine whose state transitions are given labels called *actions*. There are three kinds of actions. The environment affects the automaton through *Input actions* which must be responded to in any state. The automaton affects the environment through *Output actions*; these actions can be controlled by the automaton to only occur in certain states. *Internal actions* only change the state of the automaton without affecting the environment.

Formally, an IOA is defined by a *state set* S, a *action set* A, an *action signature* Z (that classifies the action set into input, output, and internal actions), a *transition relation* $R \subseteq S \times A \times S$, and a set of *initial states* $I \subseteq S$. We mostly deal with *uninitialized IOA* for which $I = S$. An action a is said to be *enabled* in state s if there exist $s' \in S$ such that $(s, a, s') \in R$. Input actions are always enabled.

Nodes communicate with each other by sending to and receiving *packets* along links. Fix a packet alphabet P. Nodes and links are modeled by IOA called *node* and *link* automata respectively. A node automaton (see Figure 1) for node i in graph G has output actions $(\textsc{Send}_{i,j}(p), p \in P)$ to send a packet to every j such that $(i, j) \in E$; it also has input actions to receive packets $(\textsc{Receive}_{j,i}(p), p \in P)$ for every j such that $(j, i) \in E$. Similarly, the link automaton for link $(i, j) \in E$ has input action $\textsc{Send}_{i,j}(p)$[2] to receive packets from i, and output action $\textsc{Receive}_{i,j}(p)$ to deliver packets to node j (Figure 1).

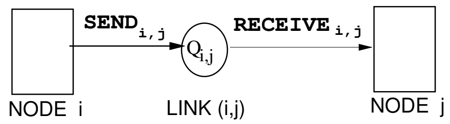

Figure 1: The rectangles represents node automata for nodes i and j with interfaces to send packets from i to j. The circle represents the link automaton from i to j whose state is captured by a queue, $Q_{i,j}$

[2] the convention for action subscripts is that the first subscript always represents the sending node and the second the receiving node.

State of link $C_{i,j}$ is a queue of packets $Q_{i,j}$.
In all states, $|Q_{i,j}| \leq L_{max}$

$\textsc{Send}_{i,j}(p)$ (* node i sends a packet *)
  Effects: Add p to the end of $Q_{i,j}$

$\textsc{Receive}_{i,j}(p)$ (* node j receives a packet*)
  Preconditions: p is the head of $Q_{i,j}$
  Effects: Remove p from $Q_{i,j}$

For time complexity, assume that any packet p placed in $Q_{i,j}$ is delivered in 1 time unit.

Figure 2: Formal Model of a Bounded Data Link

Node automata can be arbitrary except that they must have *finite* state sets, and have the appropriate interface actions to send and receive packets. We assume that each link is a FIFO queue with *bounded storage*. More formally, the state of the link automaton for link (i,j) is a queue of packets $Q_{i,j}$, that is restricted to store no more than L_{max} packets. A formal specification of a *bounded Data Link* is in Figure 2. We use a bounded model because not much can be done with *unbounded* links in a stabilizing setting [DIM91], and real links are bounded anyway. We describe why we believe this is a good model for many real networks at the end of this section.

Define an *network automaton* for graph $G = (E, V)$ as the composition of node automata for each $i \in V$ and link automata for each edge $(i,j) \in E$. *Composition* [LT89] produces a composite state machine; input and output actions of the same name are performed simultaneously. Thus when node i performs a $\textsc{Send}_{i,j}(p)$ output action, the link between i and j performs a simultaneous input action (also $\textsc{Send}_{i,j}(p)$) to store packet p. The state of the composition is the cross product of the states of every node and link automaton in the network automaton.

When an IOA "runs" it produces an execution. An *execution fragment* is an alternating sequence of states and actions $(s_0, a_1, s_1, \ldots)$, such that $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. An execution fragment E is fair if any internal or output action that is continuously enabled eventually occurs.[3] An *execution* is an execution fragment that begins with a start state and

[3]The IOA model actually specifies fairness in terms of equivalence classes; for this paper we assume each

is fair.

We express the correctness of an automaton using a set of *legal executions* LE as in [DIM93]. Let LE be a set of executions of some automaton A. Then we say that automaton A stabilizes to LE if every execution of A has some suffix that is contained in LE. The *legal states* are the states that occur in legal executions. All the automata we will design in this paper will be *uninitialized IOA* whose set of initial states I is identical to its state set S. We do so by not specifying initial node or channel states. It should be clear that the intuitive concept of self-stabilization is captured by the stabilization of an uninitialized automaton to a set of legal executions.

For time complexity, assume that every internal or output action that is continuously enabled at a node occurs in 1 unit of time. Thus node processing takes 1 time unit. However, we assume any packet stored on a link is delivered in 1 time unit. We say that A *stabilizes* to *LE in time t* if every execution of A has a suffix that occurs within time t and is contained in LE. The *stabilization time* from A to B is the infimum across all t such that A stabilizes to B in time t.

The time complexity assumption for messages is reasonable for links, such as fiber links, in which packets are not queued on links. The assumption is completely unrealistic for channels that act like queues; a simple example is when the link is really a "network" which may consist of internal switching nodes that can queue packets. However, since we expect our protocols (such as token ring, broadcast on trees, and resets) to be used over networks with the former type of link, we believe this assumption is reasonable.

The time complexity assumption also seems to imply (see description of algorithms later) that each node has to send a stabilization message every step. In particular, this seems to follow because we have made the time complexity for nodes to send messages the same as the time complexity for a message to travel on a link. We choose this for simplicity in order to avoid having two parameters for the two times. A similar (but slightly more messy) analysis of the algorithms can be carried out in which the time to perform an internal node action is bounded by some parameter t_n. The general model would allow stabilization messages to be sent at any reasonable interval, and would provide the usual tradeoff between message overhead and stabilization time.

There are several other methods of calculating time complexities for stabilizing protocols. These include methods in which messages in which time complexity is measured in terms of rounds in which every processor takes a step. Our time complexity measure is not directly comparable to these other measures.

action is in a separate class.

Why Use Bounded Links? In a stabilizing setting, if a link can store an unbounded number of packets, it is impossible to produce solutions (with bounded stabilization time) for most non-trivial tasks [DIM91]. Moreover, real links are bounded.

In other work, we have modeled bounded links as unit capacity data links (UDLs) that can store at most one packet at any instant. A UDL [Var93] is a model of a reliable Data Link protocol that delivers one packet at a time. The UDL model is appropriate for routing (and other) protocols that use an underlying reliable link protocol. However, many real protocols that work over very low error rate links (e.g., FDDI, ATM, Frame Relay [Tan93]) do not use an underlying Data Link protocol. Such links store a bounded number of packets because the link sender and receiver are *synchronous*. The receiver removes packets as fast as the sender inputs packets. However, there is no common clock for the entire network; *node processing is still asynchronous*.

We can model this using an asynchronous model like IOA if we only consider executions in which there are no more than L_{max} packets on each link in each state. In our bounded model we assume that any stored packet is delivered in constant time. For example, suppose the minimum packet size is 20 bytes, nodes transmit at 100 Mbit/sec and links are up to 10 miles long. Then $L_{max} = 30$ and (assuming speed of light limitations) any stored packet is delivered in 50 usec. Both numbers are constants that depend only on the maximum length of a link.

3 Counter Flushing on Rings

To illustrate counter flushing, we first show how counter flushing can be used in rings. Our protocol is a message passing version of a shared memory protocol presented in [Dij74a]. We have a leader node 0 called the root. Nodes $0 \ldots n-1$ are arranged in a ring topology such that there is a directed link $(i, i+1)$ for $i = 0, \ldots n-1$. Assume that addition on process indices is always implicitly $\bmod n$ so that $n - 1 + 1 = 0$.

We wish to do mutual exclusion on this ring topology. In a ring, without the need for stabilization, this can easily be done by sending a special *Token* packet round the ring. Each node i would then have a flag which would be set to true when a token arrives at i, and is set to false when node i sends its token to node $i+1$. As long as we start in a state where there is exactly one token, this protocol will maintain a mutual exclusion property: no more than one node can have its token flag set to true in any state.

However, in a stabilizing setting, the token protocol can deadlock. Clearly, we can start the token protocol in a state that does not contain a token. A simple way to avoid deadlock is to have nodes retransmit packets. But this introduces the possibility of a node receiving duplicates during each cycle. Thus we change the state of each node i (and each packet) to have a counter instead of a flag. A node with counter value c can identify a packet numbered c as being a duplicate. This is analogous to the use of sequence numbers in network protocols [Tan93].

However, counters cause new complications. In the initial state, the counter values may be arbitrary. Let c_{max} be the maximum number of counters that can be stored in the network in the initial state. Because of the initially bounded model, $c_{max} = |E|L_{max} + |V|$. We will show that if the size of the counter space is greater than c_{max}, then the protocol stabilizes in time proportional to the network diameter. The underlying constant is small, typically equal to 3. We will prove that every execution reaches a state in which the root has a *fresh* value that is not present in the network; then we show that any execution starting with a "fresh" value will reach a legal state. The fresh value propagates through the network "flushing" old values much like cleaning fluid moving through a messy drain.

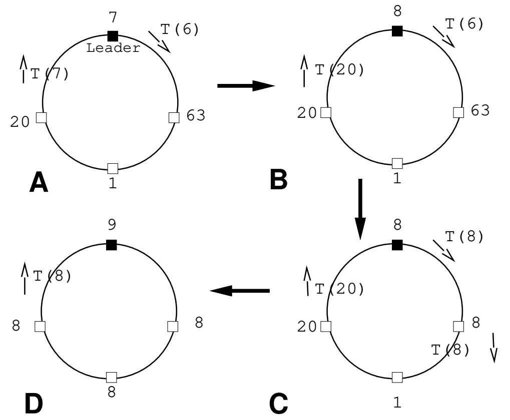

Figure 3: Progress to Stabilization in a Token Ring with Counters. Starting with an arbitrary state (**A**) we reach a state in which the root has a fresh value (**B**). The fresh value (i.e., 8) moves round the ring (**C**) until we reach a legal state (**D**)

In Figure 3 the lower left hand corner (configuration **D**) describes a good global state of our protocol. There are four nodes which we call North, East, South and West and all packets travel clockwise around the ring. North is the root. Each node other than North has a counter value of 8 and there is a token carrying the value 8 in transit from West to North.

Each node periodically retransmits its counter value in a token packet downstream. Thus mere receipt of a token packet is not enough for a node to assume it has the token. Instead, when any node i receives a token from its upstream neighbor, node i does the following. If i is not the root and the counter value in the token (say c) *is different* from the counter stored at node i (i.e., c_i), then node i assumes it has received a *valid token* and sets c_i equal to c; if

$c = c_i$ and i is not the root, i ignores the received token. If i is the root, however, a different rule is used: if the counter c in the token *is equal* to the root's local counter, then the root assumes it has received a *valid token*, and increments its counter value modMax; if $c \neq c_i$, then the root ignores the received token.

Consider legal configuration **D** of Figure 3. In this state all token packets on links have a counter value 8, and the counter values at all nodes except North is also 8. The counter value at North is $9 \neq 8$. In that case, we say that North has the token. Eventually North will transmit a token packet containing 9. When this packet reaches East, East sets its counter value to 9. This process continues with the token moving clockwise until West receives the token and transmits it to North. North chooses a new value (10) and the cycle continues.

In legal states the ring can be partitioned into two bands. The first band starts with the root and continues up to (but not including) the first counter value (either in a token packet or at a node) whose counter value is different from that of North. The remainder of the ring (including links and nodes) is a second band containing a counter value different from North. The valid token is at the end of the first band.

The protocol stabilizes to legal states regardless of *initial values of node and packet counters.* Assume c_{max} (the number of distinct counter values stored in nodes and links in the initial state) is less than the counter size Max. For example, with a 1000 node ring transmitting at 100 Mbp/s and assuming 10 mile links, $c_{max} = 31,000$. If we use a 32 bit counter, then $Max = 2^{32} > c_{max}$ for even the largest size rings that occur in practice. The stabilization argument is illustrated in Figure 3. We provide a formal argument in Section 6. For now we sketch an informal argument that illustrates the three essential features of the counter flushing argument (increment liveness, freshness, and flushing.)

1. In any execution, North will eventually increment its counter: Suppose not. Then North's value will move around the ring until North gets a token with a counter value equal to its own.

2. In any execution, North will reach a "fresh" counter value not equal to the counter values of any other process: (see Figure 3, configuration **B**). In the initial state there are at most (say) c_{max} distinct counter values. Thus there is some counter value say f not present in the initial state. Since North keeps incrementing its counter, North will eventually reach f; in the interim no other process can set their counter value to f since only North "produces" new counter values.

3. Any state in which North has a fresh counter value f is eventually followed by a state in which all processes have counter value f: (see Figure 3, configurations **C,D**). The value f moves clockwise around the ring "flushing" any other counter values, while

North remains at f.

Define a *round trip delay* to be equal to $2N$ time units (i.e., the time it takes for a packet to travel around the ring with a unit delay at each node and link.) The worst-case stabilization time of this protocol is equal to 3 round trip delays.

First consider a modification to the protocol in which the root chooses a new counter value *randomly* instead of incrementing the old value. Assume that $Max \gg c_{max}$ (i.e., counter size is much greater than the maximum number of stored packets. With very high probability (i.e., $1 - c_{max}/Max$, roughly 2^{-16} for our ring example), the root picks a "fresh" value after its first opportunity to change its counter. The protocol stabilizes within 2 round trip delays times with high probability. We call this *randomized counter flushing.*

However, simple *deterministic* incrementing also guarantees a worst case stabilization time of 3 ring round trip delays. Here is the intuition. Consider an execution with initial state s_I and some state s_F that occurs 1 round trip delay later. Let the counter value of the root in s_I and s_F be $c(I)$ and $c(F)$ respectively. In one round trip delay, there is enough time for information from the root to "flow" through the entire ring. Thus all node counters in state s_F must have been "produced" by the root since the execution began. More formally, in s_F, all counter values are in the range $[c(I) \ldots c(F)]$. If $c(F)$ is fresh, we are done (see stabilization argument above) 1 round trip delay later. Otherwise, the next root increment will cause a fresh value because $c(F) + 1 \notin [c(I) \ldots c(F)]$. (This last fact follows because the root will increment at most once for every packet received, and can receive at most $c_{max} < Max$ packets in the interval $[s_I, s_F]$.) But the next increment will happen after at most one round trip delay.[4]

The formal code for our stabilizing token passing protocol is in Figure 4. Our protocol is a message passing version of the shared memory protocol [Dij74a]. One of our contributions is to prove that the stabilization time is equal to 3 ring delays, using our model of time complexity[5], which we believe is realistic. But our main contribution is abstracting the mechanism and applying it to other examples, as we show below. A formal proof of correctness and stabilization is deferred to Section 6.

We note that token passing protocols are widely used in Local Area Networks in order to regulate access to the network. Existing token passing protocols recover from lost tokens using global timers that are refreshed whenever a token is seen. In the IBM token ring [Tan93], the monitor (i.e., root) uses a timer that is set to the longest possible delay it can take for a token to traverse the ring. When this timer expires, the monitor reinitializes the ring. Thus the

[4]A more careful accounting shows that 2 round trip delays suffice for stabilization. We prefer to use 3 round trip delays because of the simplicity of the proofs.

[5]There are no time complexity results in [Dij74a]

recovery time of the IBM token ring protocol *is proportional to the worst-case delay around the ring.* The recovery time of a token passing protocol based on counter flushing (see [Cos96] for details) is *proportional to the actual delay around the ring, which can be an order of magnitude faster* than the worst-case delay.

3.1 Further Applications of Counter Flushing on Rings

Counter flushing on rings can be applied to two more interesting settings: request-response protocols, and alternating bit protocols between two nodes.

Request-Response Protocols: Suppose a leader node wishes to periodically send a *Request* packet to a set of network nodes. The responders must each send back a *Response* packet before the sender sends its next request. In order to properly match responses to requests, the sender numbers [Var93] each request with a counter. Responders only accept *Request* packets with a number different from the last *Request* accepted. After accepting a *Request* the responder sends back a *Response* with the same number as the *Request.* The sender retransmits the current *Request* till it receives each matching *Response* with the same number. After all matching *Response* packets arrive, the sender increments its counter and starts a new phase. The protocol will work correctly if $Max > c_{max}$ and the links are FIFO (or guarantee the 'flushing" property in some other way.)

Data Link Protocols: The token passing protocol in Figure 4 can be adapted to send packets reliably between a sender and receiver by having each token packet carry a piggybacked data packet. It is important to compare this with the elegant stabilizing Data Link protocol of Afek and Brown [AB93]. They use bounded length counters of size greater than 2, but such that the sequence of counter values used is aperiodic. A trivial corollary is that for a pair of nodes connected by a pair of unidirectional links, it suffices to use a counter whose size is larger than 1 plus the maximum number of outstanding packets. They also suggest the use of a random sequence instead of an aperiodic sequence.

However, Afek and Brown's result is confined to Data Link protocols between a pair of nodes and to rings. The paradigm has not been extended to trees or general networks as we do below. Also the randomized equivalent of Afek-Brown's protocol uses randomized sequences instead of the Random-Increment function; the expected stabilization time of their protocol is shown to be $O(c_{max})$ round trip delays for large values of *Max*, while our stabilization time is only approximately 2 round trip delays.

A token packet is encoded as a tuple $(Token, c)$ where c is an integer in the range $0..Max$
The state of each node i consists of an integer $count_i$ in the range $0..Max$.
The root has an additional flag $token\_expected_0$.
Assume there are n nodes numbered from 0 to $n-1$.
All addition and subtraction of process indices is mod n.
All addition and subtraction of counters is mod Max.

$Finished(i)$ (* boolean function used by action routines below *)
Always true for all nodes other than node 0
$Finished(0)$ is true if and only if $token\_expected_0 = false$

ROOT_START (* Node 0 is considered the root or leader of ring *)
Preconditions:
$Finished(0) = true$
Effects:
$count_0 = count_0 + 1$ (* increment root counter *)
$token\_expected_0 = true$

RECEIVE$_{n-1,0}(Token, c)$ (* node 0 receives token from node $n-1$ *)
Effects:
If $c = count_0$ then (* token counter matches node counter *)
$token\_expected_0 = false$ (* node 0 treats this packet like an ack*)

RECEIVE$_{i-1,i}(Token, c), i \neq 0$ (* node i receives token from clockwise neighbor node i-1*)
Effects:
If $c \neq count_i$ then (* token counter differs from node counter *)
$count_i = c$ (*set value to counter in token packet*)

SEND$_{i,i+1}(Token, c)$, (* node i sends token to clockwise neighbor node i + 1*)
Preconditions:
$c = count_i$ (* counter of token matches node counter *)

For any node, a SEND$_{i,i+1}$ action will occur in 1 unit of time
starting from any state.

Figure 4: Code for node processes in a token ring

4 Counter Flushing on Trees

In the last section, we described counter flushing on a ring topology using a mutual exclusion protocol that is essentially sequential. By contrast, in this section, we describe a broadcast protocol on a tree which exhibits considerable parallelism. In this problem, we have a root node 0 that wishes to broadcast a sequence of values to every node in the network. Correct executions of the protocol can be partitioned into an infinite number of cycles: in cycle M the root must send the packet corresponding to M exactly once to all network nodes. Cycle M begins after cycle $M-1$ ends. In order to detect when the current broadcast cycle has terminated the root needs to obtain feedback from the other nodes. Thus the problem is often called Propagation of Information with Feedback (PIF).

We model the sequence of values that the root wishes to broadcast by having the root have access to a function f that computes the next value to be sent as a function of the previous value sent. In a more general setting, the values could be supplied by some external application.

We assume a leader/root node 0 and a spanning tree rooted at node 0, such that each node i has a parent variable $parent(i)$ that points to its parent in the tree. Without stabilization, it is easy to solve this problem using the protocols due to Segall and Chang [Seg83, Cha82]. When the root finishes broadcasting a previous value, it chooses a a new value using the function f. It then sends a token packet containing the new value to all its children; other nodes accept new values only from their parents, upon which they send the value to their children. When a leaf of the tree gets a new value, it sends an ack up to its parent. Nodes other than the root send an ack up to their parents when they have received acks from all children. When the root receives an ack from all children, the root starts a new cycle by choosing a new value. Clearly this protocol can deadlock if initialized in a state where the root is expecting acks from its children, but the children do not send any further acks.

To make the protocol stabilizing, we tag each packet sent (and each value stored) with a counter. When sending a new value, the root chooses a new counter value. Node i accepts a new value only when it is tagged with a *different* counter value from the counter stored at node i. Node i accepts an ack only when the counter in the ack is *identical* to node i's counter. Adding counters and checks also allows us to periodically retransmit *Token* packets to avoid deadlock.

The code is given in Figure 5. For simplicity, we do not encode "Acks" separately but just have children send $(Token, c, v)$ packets to their parents as acks, where c and v are the counter and value respectively at the sending node at the instant the packet was sent.

Figure 6 shows a legal state of the protocol where a new value x is being broadcast to replace the previous value v. The new counter tag for x is 13 while the counter tag for v was

We assume all counters are integers in the range $0..Max$ and all values are drawn from some domain V.
A token packet is encoded as a tuple $(Token, c, v)$ where c is a counter and v is a value.
The state of each node i consists of:
a counter $count_i$, a boolean flag $token\_expected_i[j]$ for each neighbor j of i and a value field v_i.
We assume that each node i has a function $parent(i)$ that points to i's parent in the tree.
We assume the root is node 0 and all addition of counters are done mod Max.

$Finished(i)$ (* boolean function used by action routines below *)
(* set to true when not expecting tokens from any children *)
Return true if i is a leaf, or if i is not a leaf and for all children k of i: $token\_expected_i[k] = false$

ROOT_START (* Root starts a new cycle of broadcasting values *)
Preconditions:
$Finished(0) = true$
Effects:
$v_0 = f(v_r)$ (* compute new value to be broadcast*)
$count_0 = count_0 + 1$ (*choose new counter value mod Max*)
For all children k of root
$token\_expected_0[k] = true$ (* set to true when expecting a correctly numbered token*)

$\text{SEND}_{i,j}(Token, c, v)$, (* node i sends token to node j *)
Preconditions:
$c = count_i$ (* counter of token matches node counter *)
$v = v_i$ (* value equal to store value *)
$j \neq parent(i)$ or $(j = parent(i)$ and $Finished(i))$ (* send to children, and to parent if finished *)

$\text{RECEIVE}_{j,i}(Token, c, v)$ (* node i receives token from node j *)
Effects:
If $j = parent(i)$ and $c \neq count_i$ then (* new counter from parent *)
$v_i = v$ (* set stored value equal to value in token packet*)
$count_i = c$ (*set local counter to counter in token packet*)
For all children k of i
$token\_expected_i[k] = true$ (* set to true when expecting a token *)
Else if $c = count_i$
$token\_expected_i[j] = false$ (* treat as a valid ack from child j*)

Any action that is continuously enabled for 1 unit of time occurs in 1 unit of time.

Figure 5: Code for Stabilizing Propagation of Information with Feedback

12. The new value x has reached the right child of the root and is in transit to the rightmost leaf node. When this leaf node gets a packet containing $(x, 13)$ it will accept the new value because $12 \neq 13$. It will then send an ack containing the counter 13 to its parent; the parent will accept this as a valid ack.

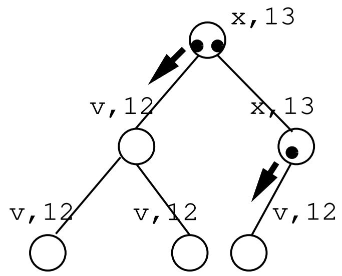

Figure 6: Using counter flushing to make Broadcast on a Tree Stabilizing. A new value x is being broadcast to replace the previous value v. A black dot indicates that a node is waiting for an ack on that link.

Suppose the counter size is greater than $Max = nL_{max}$. Then the counter flushing argument guarantees that this protocol will enter a legal state in $4h + 2$ time (h is the tree height) regardless of the initial state. Once it stabilizes, the protocol correctly broadcasts subsequent values generated at the root. We defer a formal proof of correctness to Section 6.

The reader may feel that because the PIF protocol works on a tree that it is possible to avoid the use of counters completely; however, it is easy to construct counterexample executions where if the counter is not used (or its size is less than Max), then the system stays in an incorrect state forever. Note that Max must once again be larger than the maximum number of outstanding counters in the initial state, which is nL_{max} where n is the number of tree nodes.

Another fairly general method for constructing stabilizing protocols is the method of local checking as described in [APV91] and [Var93]. However the PIF protocol of Figure 5 is not locally checkable and so the earlier technique is not applicable. In a good state of the PIF protocol there can be at most two values present in the tree, the value currently being propagated and the old value that is still present in the lower limbs of the tree. Thus in a good local state it is possible to have a parent have counter c and the child have counter $c' \neq c$. But in that case we can construct a bad global state in which each child of the root has a different counter value but each pair of neighbors appears to be in a good state locally. Thus the protocol of Figure 5 is not locally checkable.

Independently, Gouda [Gou94] used the concept of observers to unify tree and ring stabilizing systems. The paper, however, uses a different proof from ours.

Applications of Counter Flushing on Trees: Propagation of Information with feedback is a specific example of a centralized total algorithm [Tel89]. A centralized total algorithm is an algorithm where each process in the network must take some decision before the initiator takes a decision event. Tel [Tel89] shows that many protocols such as PIF, Finn's Resynch Protocol [Fin79], and distributed infimum[6] are all examples of Total algorithms. Tel also shows that PIF can be used to solve any total algorithm. Thus the stabilizing PIF protocol described in Figure 5 appears to be important because it appears to offer a stabilizing solution to many problems that require total solutions. We note that another interesting application of the stabilizing PIF protocol is for topology update. For example in the Autonet network [MAM+90], topology distribution is done over a tree.

5 Counter Flushing in General Graphs with a Spanning Tree

We broaden the scope of counter flushing to consider general graphs. However, we continue to assume that we have a root r that is the root of a BFS spanning tree. Besides links from parents to children we now also have cross links that are not part of the tree. So far we only seen how to use counter flushing to flush tree links. We now extend the paradigm so that the use of a fresh counter value at the root will flush all links, both tree and cross links.

The basic idea is simple. As before a node i only accepts a new counter value c from its parent, and waits till it gets tokens from its children (numbered with c) before it sends a token up to its parent. However, in addition, i sends a token packet on any cross links it is part of, and waits to get a token (numbered with c) on every link before it sends a token to its parent.

The only difficulty is deciding how to reply to token packets received on cross links. Before we see what the problems are, we introduce an application for this general counter flushing paradigm. Suppose we have an underlying protocol P and suppose that the root may periodically get a request to reset protocol P. We stipulate that at the point the reset procedure terminates, the state of the underlying protocol P is reset to some successor of a legal initial state of P. To do so, at some point during the reset procedure i) each node i must locally reset its Protocol P state ii) Define the *reset interval* of a node to be the interval from the time a node is locally reset until the reset procedure terminates. Then for any pair of neighbors i, j the sequence of packets received by node j in j's reset interval must be a proper prefix of the sequence of packets sent by node i during i's reset interval.

In Figure 7, node i has received the counter value (5) corresponding to the current reset

[6]this can be described roughly as calculating a bound on the minimum of a set of values stored at network nodes and links

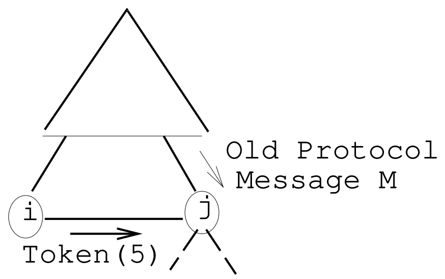

Figure 7: Reason for delaying responses to token packets received on cross links.

and has sent a token packet containing 5 to j. Node j has not received information on the current reset and has an "old" protocol packet in transit from its parent. Suppose node i's token packet reaches node j first and node j sends back a token immediately (but without changing its counter value or initializing protocol P). Then node j can subsequently receive the "old" protocol packet and send another "old" protocol packet to node i. Thus we could have a packet sent before node j was reset being received by node i after node i has reset, an error.

It may appear that a simple solution is for node j to reset itself locally (and accept the new counter value) when it receives the token packet on the cross link from node i. But that causes the entire counter flushing paradigm to break down. This is because if a node accepts counters on cross links to its neighbors then in the initial configuration we could have a cycle of nodes each having different values which results in a form of livelock, where the counters move around in the cycle. This problem can occur for instance in the protocol proposed by Katz and Perry [KP93]. Katz and Perry resolve this livelock problem by having each token packet carry a counter *and* a list of visited nodes; a token packet is dropped when it visits a node in the visited list. While this solution works, it increases message and time complexity.

The livelock problem disappears if nodes only accept counter values from their parents as we have done in the last subsection. To solve the problem we referred to in Figure 7, we do two things. First, we can tag all protocol P packets with the counter at the sending node; we discard a protocol P packet with a counter that does not match the receiver's counter. While this solution eliminates the problem in Figure 7 because the "old" packet will have a different counter value from that of node i, it introduces another problem. Suppose node i sends a protocol P packet to j after node i resets, but the packet is received before j resets. Then if we simply check the packet tag, the packet will be dropped at j. One might consider buffering the packet at j if the counter tag in the packet is "greater" than the counter at j; however, defining one counter to be greater than another is fraught with complications if the counters are of bounded size.

Instead we have each node j *delay* responding until the local counter at node j is equal to the counter of the token packet received. Thus in Figure 7 when j receives the token packet from i numbered 5, node j does not send a token numbered 5 back to node i, until node j has also received a token packet numbered 5 from its parent. In the meanwhile, node i will keep retransmitting a token packet numbered 5 to j. Node j will ignore these packets until it, too, has the same counter value of 5. It will then send a token packet back to i with number 5.

We also do not allow protocol P to send packets at node i if node i is waiting for a token packet on one of its links. This implies that (in good executions) any packet sent by i after i has locally reset is sent after j is at the same counter value as i; thus this packet will be accepted by j.

The formal code for this protocol is described in Figure 8. Once again, we defer the proof of correctness and stabilization to Section 6. We will show there that the reset protocol stabilizes in three round trip delays.

5.1 Comparison with Other Reset Protocols

To construct a full-fledged reset protocol, we need to augment the protocol described so far to allow any node to make a reset request. This is done as follows. Each node has a reset request bit that is set when the node gets a reset request, or when it has received a *Request* packet from its children. When a node's request bit is on, it periodically sends a *Request* packet to its parent. When the root gets a *Request* packet, the root treats it like a REQUEST_RESET action. On doing a local reset a node clears its request bit; each node i also ignores reset requests and *Request* packets while $Finished(i) = false$.

The resulting reset protocol is similar to a stabilizing reset protocol due to Arora and Gouda [AG94] but has some important differences. First, the protocol [AG94] is based on a shared memory model and thus only requires node counters of size 2. In a message passing model, where each link can store L_{max} counter values, we believe that larger counter values, as in our protocol, are necessary. A second difference between our protocol and the one in Arora-Gouda is the use of "delayed acks" and flushing of cross links. This is unnecessary in [AG94], because protocol P is modified so that a node does not read the state of its neighbors unless they have the same counter value. This is possible in a shared memory model but not in a message passing model.

There is also the stabilizing reset protocols of [APV91] which is in turn based on the non-stabilizing reset protocol of [AAG87]. However, this protocol takes $O(n)$ time to stabilize which is slower than our reset protocol or the Arora-Gouda protocol. [AKM+93] suggests making

A token packet is encoded as a tuple $(Token, c)$ where c is an integer in the range $0..Max$
The state of each node i consists of:
an integer $count_i$ in the range $0..Max$, and a flag $token\_expected_i[j]$ for each neighbor j of i:
$token\_expected_i[j]$ is always *false* if $j = parent_i$
We assume $parent(i)$ points to i's parent, the root is node 0, and addition of counters is mod Max.

$Finished(i)$ (* boolean function used by action routines below *)
(*set to true when not expecting tokens from any non-parent links *)
Return true if for all neighbors $token\_expected_i[k] = false$

ROOT_START (* root receives request to reset Protocol P *)
Effects:
if $Finished(0)$ then (* ignore request if finishing current reset *)
$count_0 = count_0 + 1$ (*choose new counter value mod Max*)
LOCAL_RESET(0) (* locally reset Protocol P *)
For all neighbors k of root, $token\_expected_0[k] = true$ (* expect an ack from all neighbors*)

SEND$_{i,j}(Token, c)$, (* node i sends token to node j *)
Preconditions: (* retransmit periodically regardless of ack bit *)
$c = count_i$ (* counter of token matches node counter *)
$j \neq parent(i)$ or $(j = parent(i)$ and $Finished(i))$ (* send to children, and to parent if finished *)

RECEIVE$_{j,i}(Token, c)$ (* node i receives token from node j *)
Effects:
If $j = parent_i$ and $c \neq count_i$ then (* new counter from parent*)
$count_i = c$ (* set value to counter in token packet*)
LOCAL_RESET(i) (* locally reset Protocol P *)
For all neighbors $k \neq j$ of i (* don't expect ack from parent *)
$token\_expected_i[k] = true$ (* set to true when expecting a token packet *)
Else if $count_i = c$ then
$token\_expected_i[j] = false$ (* treat as a valid ack from neighbor j*)

RESET_FINISHED$_0$ (* root reports finishing of reset *)
Preconditions:
$Finished(0)$

Protocol P packets are only sent at node i when $Finished(i)$ is *true* and are tagged with $count_i$.
A protocol P packet M received at node i is relayed to the application iff the tag of M is equal to $count_i$.
Any action that is continuously enabled for 1 unit of time occurs in 1 unit of time.

Figure 8: Simple Reset Protocol using Counter Flushing

this protocol faster by running it over a spanning tree, but in that case much of the complexity of that protocol is not needed. The fast and lean reset protocol of [IL94] does a reset in constant space. We believe that a 32 bit counter is adequate for most networks, and hence the requirement for logarithmic space in our protocol is not a problem for practical protocols. Our reset protocol is also much simpler.

6 General Proofs

In this section, we present our proofs of stabilization and correctness for the three protocols (Token Ring, PIF, and Reset) described in Figures 4, 5, and 8 respectively. The three protocols seem very different, work on different topologies, and have different objectives. Despite this, we will describe a uniform stabilization proof that will apply to all three protocols.

We describe the legal states of all three protocols as uniformly as possible. However, we are sometimes forced to distinguish between the Ring System (Figure 4) and what we call the Tree Systems (the PIF and reset protocols, Figures 4 and 8). There is also a small price to be paid for uniformity of proof: we added an extra ROOT_START action to the Ring System that is strictly not needed; we also used the ROOT_START action name in the Reset Protocol (to denote the action that initiates a Reset) instead of a more mnemonic name.

Our proof is structured in four subsections: in Section 6.1 we define some useful terminology; in Section 6.2 we describe the legal states of all three protocols; we prove that all three protocols stabilize quickly to the legal states and legal executions in Section 6.3; finally, in Section 6.4 we show that all three protocols exhibit the desired properties (that we have intuitively described before) in legal executions.

6.1 Definitions

We have already defined the parent of a node for tree systems. For a ring, we define the parent of node i to be node $i-1$. We define the *parent path* of a node i to be the sequence of nodes $i_0, i_1, i_2, \ldots, i_l$ such that $i_0 = 0$, $i_l = i$, and the parent of i_m is equal to i_{m-1} for $m = 1, \ldots, l$. We define the links in a parent path $i_0, i_1, \ldots, i_l$ to be the links $(i_0, i_1), \ldots (i_{l-1}, i_l)$. Notice that the links in the parent path are the links directed "downwards" from the root, and do not include any "upward" or "cross" links.

We define the counter at a node i to be $count_i$. We will often use *root counter* to denote $count_0$, the counter at the root node 0. We say that the counter in packet m is c if m is of the form $(Token, c, *)$. We say that packet m' is behind packet m in link (i, j) in some state s if

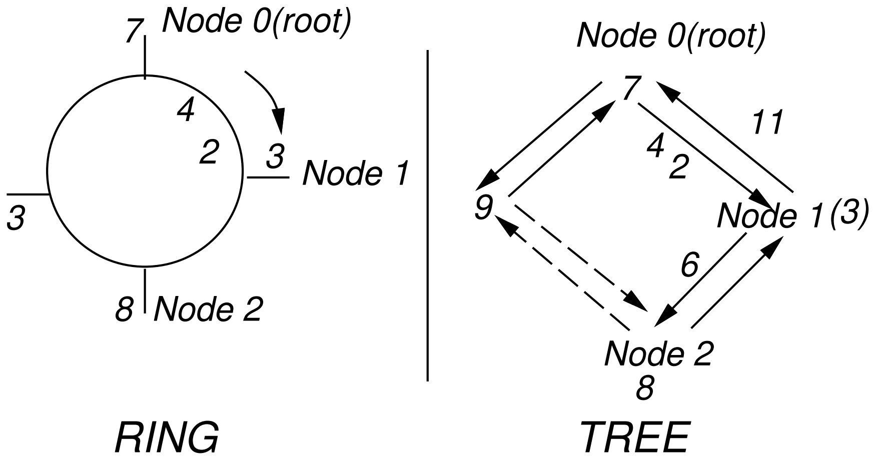

Figure 9: Examples of parent paths and upstream counters in a Ring and Tree. The numbers at nodes represent node counters and the numbers attached to a link represent the counter in a $(Token, *)$ packet stored on that link.

m and m' are both stored in $s.Q_{i,j}$ and m is closer to the head of $Q_{i,j}$. Recall from Figure 2 that $Q_{i,j}$ models the queue of packets that represent undelivered packets on link (i, j).

In order to describe the legal states we need to define the notion of counters upstream from a node or packet m. Intuitively, these are counters stored on the parent path that leads to m.

Let s be any state of the tree or ring systems. Formally, we define the *counters upstream* from node m in state s to be the set of counters in all nodes (including m) and links in the parent path of m in state s. If a packet m is stored on link (i, j) in state s, then the *counters upstream* from packet m is the union of the set of counters in packets behind m in link (i, j), together with the set of counters upstream from node i in state s. We will often refer to the counters upstream from m without reference to state s if it is clear by context what state s is.

Figure 9 gives examples of these definitions for a ring system (left) and tree system (right). Notice first that the ring system has a sequence of unidirectional links oriented clockwise in the picture. However, the tree system has a pair of unidirectional links between every pair of neighbors. In the case of the reset protocol, the tree system may include cross links (shown dashed) between neighbors such that neither is the parent of the other. The figure on the left shows a state of the ring system, and the figure on the right shows a state of the tree system. The numbers at nodes and links represent node and packet counters. Thus in both states, link $(0, 1)$ has two stored packets, the first with counter 2 and a second packet behind the first with counter 4. In both states, node 2 has node counter 8.

The parent path of node 2 is the sequence $0, 1, 2$ in both pictures. The links in the parent path of node 2 are $(0, 1)$ and $(1, 2)$ in both pictures. In the state on the left of Figure 9, the counters upstream from node 2 is the set $\{8, 3, 2, 4, 7\}$. In the state on the right, the counters upstream from node 2 is the set $\{8, 6, 3, 2, 4, 7\}$. Notice that the set does not contain 11 as the packet containing 11 is not in the parent path of node 2. In both states the counters upstream from the packet containing counter 2 is the set $\{4, 7\}$.

6.2 Legal State Definitions

Before we describe the legal states, we first describe a one-band property that holds in legal states. The one band property is a useful stepping stone in proving stabilization results. Intuitively, the system satisfies the one-band property if there is a continuous region starting at the root and consisting of nodes and packets with counters equal to the root counter. There are also some additional predicates.

The one-band predicate is illustrated in State **C** of Figure 3 for a ring system. Notice that there is a band of counters starting from the root extending to East and a token packet on the link from East to South, all of which have counter equal to 8. Notice that the remaining counters in the ring are not equal to 7, which is what we would expect in a legal state. Thus State **C** satisfies the one-band predicate but not the legal state predicate, both of which are defined below. Notice that the counters below the band can be arbitrary. Figure 6 illustrates a legal state for the PIF protocol. Notice a single band of values equal to 13 starting from the root and extending to the right child of the root. Even if the other node counters were arbitrary, this would suffice to satisfy the one-band predicate. We proceed formally.

Definition 6.1 *We say that a state satisfies the* one band *property if the following five predicates hold in that state:*

- **O1:** *If there is packet or node m with counter equal to the root counter, then all counters upstream of m are equal to the root counter.*
- **O2:** *If Finished(0) is true, then all counters are equal to the root counter.*
- **O3:** *(Tree Systems only) If a packet m is on link (i, j) with counter equal to the root counter and j is the parent of i, then Finished(i) is true and $count_i = count_0$.*
- **O4:** *(Tree Systems only) If $token\_expected_i[j] = false$ and $count_i = count_0$ and i is the parent of j, then Finished(j) = true.*
- **O5:** *(Tree Systems only) If $token\_expected_i[j] = false$ and $count_i = count_0$, then all counters in link (j, i) and $count_j$ are equal to the root counter.*

The following lemma states that once the one band property begins to hold in any execution of any system, it continues to hold.

Lemma 6.1 *The one band predicate is a stable property for all three systems.*

Proof: Easily checked by checking all possible transitions from such a state. Details can be found in [Var98]. $\square$

We can now define legal states to to be those in which the one band property holds and all counters not equal to the root counter are one less than the root counter. For tree systems, we also require that if a node counter is not equal to the root counter, then it is not expecting any acknowledgements. For PIF systems, we also require a value correspondence property. In the following definition recall that addition and subtraction of all counters is always implicitly $\bmod Max$.

Definition 6.2 *We say that a state s of any of our three systems is a legal state if:*

- **L1:** *s satisfies the one band property*
- **L2:** *Any counter that is not equal to the root counter c is equal to $c - 1$.*
- **L3:** *(Tree Systems only) If $count_i$ is not equal to $count_0$, then $Finished(i) = true$.*
- **L4:** *(PIF systems only) For all j, k where j and k can either be packets or nodes, if the counter associated with j is equal to the counter associated with k, then the two associated values are the same.*

We will refer to **L4** as the *value correspondence* property. The ring and reset systems trivially satisfy value correspondence in all states.

Lemma 6.2 *The legal state predicate is a stable property for all three systems.*

Proof: Easily checked by checking all possible transitions from such a state. Details can be found in [Var98]. □

6.3 Stabilization Proof

Define a *home* state for all systems to be a state in which all counters in nodes and packets are equal and $Finished(i) = true$ for all nodes i. The following lemmas are easy to check using the definitions.

Lemma 6.3 *Any home state that satisfies value correspondence is a legal state,*

Lemma 6.4 *Any state that satisfies the one band property and has $Finished(0) = true$ is a home state.*

Formally, define a *fresh* state for all systems to be a state in which all counters in packets and nodes other than the root are *not* equal to the root counter and $Finished(0) = false$. The following lemma is easy to check from the definitions.

Lemma 6.5 *Any fresh state satisfies the one band property.*

Let the height h of the system be the maximum length of a parent path. Clearly $h = n - 1$ for the ring. Let R, the round trip delay of a system, be $2h + 2$ for the ring and $4h + 2$ for the tree systems. Intuitively, R represents the maximum time for information sent by the root to causally flow to all nodes in the system and then flow back to the root.

The following and subsequent lemmas are stated in terms of time complexity results. They can be translated to liveness results (which do not require the time complexity assumptions we made in our model, and only require standard fairness assumptions) by replacing "within time X" by "eventually"

Lemma 6.6 *Within R time of any state s_I, either the root counter will change or the protocol will enter a home state.*

Proof: Let s_F be a state after s_I such that R units of time have elapsed from s_I to s_F and such that $count_0$ has not changed in the interval $[s_I, s_F]$. We will show that s_F is a home state. Let h_i be the length of the parent path of node i in all three systems. It is easy to show by induction that within time $2h_i$ of s_I each node with height h_i will set $count_i = c$ and $count_i$ will remain unchanged till state s_F. Intuitively this is because any packet on a link is delivered within 1 time unit, each node accepts any value sent to it by its parent, and each node retransmits a new counter value to its children in 1 unit of time.

Thus in time $2h$, all nodes will have their counter values equal to c and will remain with this value to the end of the interval. For the ring system, in time $2h + 2$, the root will receive a packet with counter equal to c and the system will enter a home state.

For a tree system, the argument is slightly longer. In time $2h + 2$, each node will receive a token packet numbered c on all its "cross" links and thus will set the *token_expected* flag to *false* for such links. Thus by time $2h+2$, all leaves l will have $Finished(l) = true$, all nodes and token packets will have counter value c, and $token\_expected_i[j] = false$ for all "cross" links (i, j). Let h'_i be the maximum length of a path from a leaf in the subtree rooted at node i to node i. Then it is easy to see, by a similar induction on the height, that within time $2h + 2 + 2h'_i$ all nodes i will have $Finished(i) = true$ and this will remain true till s_F. Thus within time $4h+2$, $Finished(0) = true$. Thus s_F must be a home state. □

Lemma 6.7 *A home state will occur within R time of a fresh state.*

Proof: Consider an execution fragment beginning with a fresh state s_I. Now within R time either the root counter will not increment (Case 1) or it will (Case 2). Consider Case 1. In that case the root counter does not increment within R time and so we must reach a home state by Lemma 6.6. So consider Case 2. Suppose the root counter increments for the first time in some state s_F that occurs within R time after s_I. We know from Lemma 6.5 that s_I satisfies the one band property as a fresh state. We know from Lemma 6.1 that all states after s_I satisfy the one band property. Thus we know that states s_F and s_{F-1} satisfy the one band property. We see from the code of all three systems that we cannot increment the root counter unless $Finished(0) = true$. Thus $Finished_0 = true$ in s_{F-1}. But by Lemma 6.4, state s_{F-1} must be a home state because s_{F-1} is fresh and has $Finished(0) = true$. Thus s_{F-1} is a home state that occurs within R time of s_I and we are done. □

We say that the root counter *wraps around* in an execution fragment $s_I, \ldots s_F$ if we have $s_J.count_0 = s_I.count_0 - 1$ for some J in $[I, F]$.

Lemma 6.8 *Any execution fragment in which the root counter wraps around must contain a fresh state.*

Proof: In state s_I there are at most c_{max} counters. Since *Max*, the modulus of the counter space, is strictly greater than c_{max}, there must be some counter value f that is not present in any node or packet in the first state s_I. Since the root counter wraps around in the interval $[s_I, s_F]$ and the root counter only changes by incrementing (mod *Max*), there must be some intermediate state in the interval $[s_I, s_F]$ in which the root counter is equal to f. Let s_J be the first such state. It is easy to see that since the value f was not present in state s_I it is not present in any state in the interval $[s_I, s_{J-1}]$. This is because, in all our systems, only the root produces new counters. Thus in state s_J only the root changes its counter value to f and sets *Finished*(0) to *false* (because the action that takes us to state s_J must be a *RootStart* action). Thus s_J is a fresh state. $\square$

Recall we defined the parent of a node i for the ring system to be node $i-1$. We define the parent link of a node i to be a link (j,i) such that j is the parent of i. We define a node or packet m to be causally connected (to the root) in an execution fragment E that ends with state s if either:

- m is the root
- m is a node and there is some state that occurs before s in which node m receives a causally connected packet on its parent link.
- m is a packet which was sent by some node i in some some state that occurs before s in which node i was causally connected.

Lemma 6.9 *Consider any execution fragment $s_I \ldots, s_F$. Suppose node or packet i is causally connected at the end of this execution fragment. Then the counter associated with i is contained in the sequence $s_I.count_0, s_{I+1}.count_0, \ldots, s_F.count_0$.*

Proof: We use induction on execution fragment length. The lemma is obviously true in the initial state of an execution fragment because only the root is causally connected and the Lemma is clearly true for root. So consider any action that extends the last state of the fragment from say s_J to s_{J+1}.

If this action is the receipt of a packet m by node i, and m is not received on a parent link then the counter of i will not change and so the Lemma remains true if it was true in state s_J. If however, it is received on a parent link, and the packet was causally connected, then after the receipt, node i is causally connected and changes its counter to the counter c associated with m. But since m was causally connected, by inductive hypothesis $c \in s_I.count, s_{I+1}.count, \ldots, s_J.count$. Thus $s_{J+1}.count_i = c \in s_I.count, s_{I+1}.count, \ldots, s_{J+1}.count$. A similar argument holds for the sending of a packet i by a causally connected node. The only other event is ROOT_START after which the Lemma clearly holds for the root and trivially holds for all other nodes whose counters remain unchanged. □

Lemma 6.10 *In any execution fragment, every node and packet will be causally connected within R time of the start of the fragment.*

Proof: Let h_i be the length of the parent path of node i in all three systems. It is easy to show by induction that within time $2h_i$ each node with height h_i will be causally connected. This follows because once a node becomes causally connected, it sends a message to each child which arrives at most 2 time units later causing the child to be causally connected. Thus all nodes will be causally connected by time $2h$ in say state s_J. Let s_F be first state after s_J in which all packets stored in links in s_J are delivered. Also, since all packets in links in s_F must have been sent after s_J, all packets in s_F are also causally connected. The lemma follows because $2h < R - 1$ for all three systems, and because packet delivery takes at most 1 time unit. □

Lemma 6.11 *If an execution fragment contains a state s which is causally connected, then state s and all subsequent states satisfy value correspondence.*

Proof: We use induction on execution length using the following inductive hypothesis. For all causally connected j, k where j and k can either be packets or nodes, if the counter associated with j is equal to the counter associated with k, then the two associated values are the same. Once all nodes and packets are causally connected the lemma follows from the hypothesis. The basis is true in the initial state as the root is causally connected. For the inductive step, if the action that extends the last state is the sending of a packet k by causally connected node i, if the counter of k is equal to some other j then the counter of i is equal to k, and thus the value of k is equal to the value of i which is equal to the value of k. A similar argument can be made for the reception of a causally connected message by a node i. □

Lemma 6.12 *Within $2R$ time of the start s_I of any execution fragment E, we will reach a fresh state or a home state that satisfies value correspondence.*

Proof: Let s_F be first state after s_J in which all packets and nodes are causally connected. From Lemma 6.10, s_F occurs within R time of s_I. By Lemma 6.11, state s_F and all subsequent states in the execution fragment satisfy value correspondence. If the root counter has wrapped around in $[s_I, s_F]$ we are done by Lemma 6.8. So assume the root counter has not wrapped around. Let $c = s_F.count_0$. Thus $c+1$ is not in the sequence $s_I.count, s_{I+1}.count, \ldots, s_F.count$. But by Lemma 6.9 all nodes and packets in states after s_J have counters in the sequence $s_I.count, s_{I+1}.count, \ldots, s_F.count$. Thus we know that as soon as the root counter first increments to $c+1$ we are in a fresh state. But we know that such a state must occur within R time of s_F by Lemma 6.6 or we will reach a home state. Since s_F occurs within R time of s_I the Lemma follows. $\square$

Lemma 6.13 *A home state that satisfies value correspondence occurs within $3R$ time of any state.*

Proof: By Lemma 6.12, within $2R$ time we reach a home state or a fresh state that satisfies value correspondence. By Lemma 6.7, within R time of a fresh state we reach a home state. Thus within $3R$ time we reach a home state that satisfies value correspondence. $\square$

So far we have not defined the legal executions of any of the three systems. By Theorem 6.14, we know that all three systems stabilize to a home state in $3R$ time, and by Lemma 6.3, we know that such a home state is a legal state. Thus it makes sense to define the legal executions of all three systems as the executions that begin with a home state that satisfies value correspondence. An immediate corollary to this definition and Lemma 6.13 is:

Theorem 6.14 *The token ring, PIF, and Reset systems all stabilize in $3R$ time.*

6.4 Correctness after Stabilization

We see from Theorem 6.14 that all three systems stabilize to legal executions in $3R$ time. We now wish to show that each legal execution will result in correct behavior for all three systems. Notice that by Lemma 6.13, we can partition a legal execution into fragments that start and end with a home state that satisfies value correspondence. We start by understanding the structure of such fragments.

Define a *fresh counter interval* to be an execution fragment[7] E such that:

[7] an execution fragment is a portion of an execution that begins and ends with a state

- The first state in E is a home state that satisfies value correspondence.
- The first action in E is a ROOT_START event.
- The second state (i.e., the state following the ROOT_START event) is fresh.
- The last state in E is the first state in E (other than than the first state) in which $\mathit{Finished}(0) = \mathit{true}$.

The value of interval E is defined to be the value of count_0 in the second (fresh) state in E. For any execution $s_0, a_1, s_1, \ldots$ we can denote an execution fragment by its first and last state indices $[I, F]$ where s_I is the initial state and s_F is the final state.

For a fresh counter interval $[I, F]$ with value c we make the following definitions:

- Let $I(j)$ be the index of the first state in $[I, F]$ such that $\mathit{count}_j = c$.
- Let $L(j, k)$ be the index of the first state after $I(j)$ which follows the sending of a packet from j to k.
- For tree systems, let $F(j, k)$ be the index of the first state such that $\mathit{count}_j = c$ and $\mathit{token\_expected}_j[k] = \mathit{false}$.
- Let $F(j)$ be the index of the first state such that $\mathit{count}_j = c$ and $\mathit{Finished}(j) = \mathit{true}$.

We now prove some simple and useful facts relating these definitions that are key to correctness.

Lemma 6.15 *For any fresh counter interval $[I, F]$, every node j and every neighbor k of j:*

- *The states $I(j)$, $L(j, k)$, $F(j, k)$ and $F(j)$ exist.*
- *$L(j, k) < I(k)$ if j is the parent of k. (i.e., a node's counter value cannot change until its parent sends it the new counter value.)*
- *In the interval $[I(j), F]$: $\mathit{count}_j = c$ (i.e., the value of a node's counter remains unchanged from the time it is initiated till the end of the interval).*
- *$I(k) < F(j, k) \leq F(j)$ (i.e., a node cannot finish until each of its neighbors is initiated.)*
- *$F = F(0)$ is a home state.*

Proof: We use the fact that any home state that satisfies value correspondence is a legal state. Since legal states are stable (Lemma 6.2), every state in the interval $[s_I, s_F]$ is a legal state and satisfies the predicates **O1**, **O2**, **O3**, **O4**, and **O5**.

- We know that *Finished*(0) $=$ *true* in s_I and s_F. Thus we know (from **O2** and **O4** applied repeatedly in s_F) that all node counters must be equal to the root counter and *Finished*(j) $=$ *true* for all nodes j. We also know that the first action causes the root counter to increment to c. Thus $I(j) = I + 1$, if j is the root. Also for all nodes j other than the root, $count_j$ is not equal to c. But in s_F, $count_j$ is equal to c. Thus there must be some first state $I(j)$ in the interval $[s_I, s_F]$ in which j first changes to c. It is is easy to see that $L(j, k)$ must exist because j will eventually send a packet to neighbor k after $I(j)$ in any execution. Also in state $I(j)$, since j changes its counter value, it is easy to see from the code that $token\_expected_i[j]$ is true. But in state s_F, $token\_expected_i[j]$ is false. Thus there must be some intermediate state $F(j, k)$ in which $token\_expected_i[j]$ first becomes false. Similarly, there must be a first state $F(j)$ in which $F(j, k)$ first becomes true for all neighbors k of j.

- In the state preceding $I(j)$, we conclude from **O1** that $count_k$ is not equal to c, and there are no counters equal to c in link (j, k). Since $L(j, k)$ is the first state after $I(j)$ in which j sends a packet numbered c on link (j, k), there can be no packets numbered c in the interval $[I(j), L(j, k)]$. Since $count_k$ was not equal to c in $I(j)$ and can only change its counter value by receiving a packet numbered c from its parent j, it follows that $L(j, k) < I(k)$.

- It is easy to see that the root cannot change its counter after s_{I+1} because the root (see code) cannot increment unless *Finished*(0) $=$ *true* and s_F is the first state after s_{I+1} in which *Finished*(0) $=$ *true*. If a node j other than the root changes its counter value after $I(j)$ to some value other than c, it means it received a counter not equal to c on its parent link. By **O1**, this implies that the root counter is not equal to c, a contradiction.

- The state s that precedes $F(j, k)$ must be (see code) the receipt of a packet with counter equal to $count_j = c$ on link (k, j). Thus by **O1**, $count_k = c$ in state s. Thus $I(k) < F(j, k)$. Also $F(j) =$ Max $F(j, k)$ over all neighbors k of j. So $F(j, k) \leq F(j)$.

- This follows immediately from **O2** in state s_F.

□

Armed with this theorem, we now show correctness separately for all three systems. Recall that any legal execution can be partitioned into fresh counter intervals. Thus to show correctness, we need only show correctness for a fresh counter interval.

Token Ring Correctness:

Theorem 6.16 *In any legal execution, at most one node has the token in any state and every node will receive the token infinitely often.*

Proof: We say that a node j has the token starting from any state s in which node j changes its counter value up to the first state after s in which node j sends a packet to $j+1$. Since it is sufficient to show correctness for a fresh counter interval within a legal execution, consider one such interval. It follows from Lemma 6.15 that $I(j) < L(j, j+1) < I(j+1)$ for $j = 0, \ldots n-1$. Thus $[I(j), L(j, j+1)]$ is disjoint for all j. Thus at most one node has the token in any state. Similarly, we know from Lemma 6.15 that $I(j)$ exists for all j and so every node j receives the token during a fresh counter interval. It follows from Lemma 6.13 and the fact that the ROOT_START event is always enabled in a home state, that the token system has an infinite number of fresh counter intervals. Thus every node receives the token infinitely often. $\square$

PIF correctness:

Theorem 6.17 *In any legal execution of the PIF system:*

- *In any state s, v_j is either equal to v_0 or the previous value of v_0.*
- *If the value of some node is not equal to v_0 in state s, there is a later state in which all node values are equal to v_0.*
- *Once a node j's value is equal to v_0, its value cannot change until we reach a state in which all node values are equal to v_0.*

Proof: First, it is easy to see that the ROOT_START event is enabled in a home state and will cause the value of the root counter to change. Thus every legal execution will have an infinite number of home states.

The first part follows from **L2** in the definition of a legal state and the fact that the counter associated with the previous value of v_0 must be $count_0 - 1$.

The second part of the theorem follows because we know from Lemma 6.13 that a home state will occur in $3R$ time after state s; in this home state $count_j = count_0$ for all nodes j. Thus by value correspondence (**L4**), $v_j = v_0$ for all j.

The third part of the theorem follows from value correspondence and the third statement in Lemma 6.15 which says that a node counter cannot change again until after the next home state. Thus by the code, its value will also remain unchanged in this interval. □

Reset Protocol Correctness: The following theorem shows that the reset protocol behaves correctly in a legal execution:

Theorem 6.18 *In every legal execution of the reset system:*

- *Once the protocol is in a home state, it remains in a home state until the next reset request, and no node will perform a local reset in this interval.*
- *Consider any reset request that occurs when the reset protocol is in a home state. Then the reset protocol will enter a home state in $O(R)$ time after this reset request and in this home state, the underlying protocol P is in a legal state.*

Proof: The first part follows easily from the code and the definitions of a home state and a fresh counter interval. Notice that when the reset protocol is in a home state, it is impossible for a node j to receive a (Token, c) packet with $c \neq \mathit{count}_j$; thus (from the code) j will never perform a local reset. We now turn to the second part of the theorem.

We know that any reset request that begins in a home state will result in the root picking a fresh counter value, say c, which begins a fresh counter interval. We know from Lemma 6.13 that within $O(R)$ time, this fresh counter interval will end. Thus from Lemma 6.15 if this interval is denoted by $[I, F]$ then there is a state $I(j)$ for each node j at which the node is initiated into the current reset computation. From the code it is easy to see that in this state, protocol P is locally reset and since count_j remains at c this means that there are no further local resets of Protocol P at node j.

To show that Protocol P is properly reset at the end of the fresh counter interval, we have to show that for any two neighbors j, k: the sequence of packets received by k from j during the interval $[I(k), F]$ is a prefix of the sequence of packets sent by j during $[I(j), F]$. Let us call the interval $[I(j), F]$ the reset interval at node j.

So consider any packet m sent by j during the interval $[I(j), F]$. From the protocol code, we know that j does not send any packet during the interval $[I(j), F(j)]$. So we can assume that m is sent after $F(j)$. Thus m will be tagged with c, the value of this fresh counter interval. Now by state F, we know from the properties of link automata, that either m will be delivered by state s_F or is stored on link (j, k) in state s_F. If m is delivered, m must have been delivered after $F(j)$ (since it was sent after $F(j)$) and hence by Lemma 6.15 it is delivered in the interval $[I(k), F]$; but in this interval, $count_k = c$ and so m is accepted. On the other hand, if (in state F) m is stored on link (j, k), we know (because the link is FIFO) that all packets sent after m are not delivered. Thus, applying this argument to all packets sent by j to k during $[I(j), F]$, we see that: if m is received and accepted, then all packets sent before m in $[I(j), F]$ are received and accepted by k; but if m is not received, then all packets sent after m in $[I(j), F]$ are not received.

All that remains is to show that any Protocol P packet m received and accepted by k in $[I(k), F]$ was sent by j in $[I(j), F]$. But if m was accepted it must have tag c. Thus m must have been sent in $[I(j), F]$; this is because, by definition, any protocol P packets sent by j in $[I, I(j) - 1]$ must have a counter value $c' \neq c$. Recall that $I(j)$ is the first state in $[I, F]$ that has $count_j = c$. $\square$

7 Conclusions

Counter flushing is a simple paradigm that has a fairly wide range of applications, and can be used over different topologies. Besides the examples discussed in this paper (token passing, broadcast, and reset), counter flushing can be used to design stabilizing protocols for deadlock detection and snapshot protocols [Var94]. The token ring protocol we described in this paper has been used [Cos96] to design a stabilizing version of the FDDI protocol. The modified FDDI protocol [Cos96] recovers from multiple tokens in less than 5.7 ms, while the existing FDDI MAC might never recover; it modified FDDI recovers from lost tokens more quickly than FDDI (0–0.36 ms versus 2.5–4.1 ms).

Our paper exploits a connection between seemingly disparate protocols such as Dijkstra's token ring protocol, Afek and Brown's Data Link protocol, and reset protocols. At one level, they can all be regarded as repeated versions of a centralized total algorithm [Tel89] in which cooperation is needed from all nodes to reach a decision; at another level, the Data Link and Reset problems can be regarded as *synchronization* problems whose correctness can be formalized in terms of a mating relation [AE83, Spi88, Var93]. The unified approach allows us to describe a general proof that applies to three rather different systems.

Counter flushing, as described in this paper, has three aspects. First, we establish the presence of a non-existent counter based on bounding the space of counters; second, we argue a liveness property that guarantees that deterministic incrementing (randomized choosing also works trivially) will lead to a unique counter in a very short time; third, we show a flushing condition to show that a non-existent counter flushes out all bad values. Thus, while the elegant papers in [Dol94, AK93] do use randomization to choose a non-existent counter, they do not need the other two conditions.

Local Checking and Correction is another general paradigm that has been used before ([APV91, Var93, AGV94]) to design and explain efficient stabilizing protocols. On a theoretical level, there are some problems for which counter flushing is applicable but local checking is not (e.g., protocols that are not locally checkable like token passing on a ring) and some problems for which local checking is applicable but counter flushing is not (e.g., synchronizers). There are also a number of problems where they are both applicable (e.g., resets, token passing on a tree). We believe that while they are both practical methods, counter flushing is simpler to implement. Local checking [APV91, Var93] requires a careful enumeration of predicates and the addition of periodic local snapshots and resets.

We have already generalized counter flushing to window washing [CV96]. We have also applied it to design a real token passing protocol in [Cos96]. Our goal is to design elegant theoretical techniques that can help design simple, effective, and practical protocols.

References

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

[AB93] Y Afek and GM Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[AE83] Baruch Awerbuch and Shimon Even. A formal approach to a communication-network protocol; broadcast as a case study. Technical Report TR-459, Electrical Engineering Department, Technion-I.I.T., Haifa, December 1983.

[AG94] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[AGV94] A Arora, MG Gouda, and G Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. In *ICDCS94 Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 424–431, 1994.

[AK93] S Aggarwal and S Kutten. Time optimal self-stabilizing spanning tree algorithm. In *FSTTCS93 Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag LNCS:761*, pages 400–410, 1993.

[AKM+93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proc. 25th ACM Symp. on Theory of Computing*, October 1993.

[APV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[Cha82] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. on Software Eng.*, 8(4):391–401, July 1982.

[Cos96] A Costello. Self-stabilization by counter flushing and window washing (M.S. thesis). Technical Report 220, Washington University, 1996.

[CV96] A Costello and G Varghese. Self-stabilization by window washing. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 35–44, 1996.

[Dij74a] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[Dij74b] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[DIM91] S Dolev, A Israeli, and S Moran. Resource bounds for self stabilizing message driven protocols. In *PODC91 Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 281–293, 1991.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[Dol94] S Dolev. Optimal time self-stabilization in uniform dynamic systems. In *6th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 25–28, 1994.

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

[Gou94] MG Gouda. Stabilizing observers. *Information Processing Letters*, 52:99–103, 1994.

[IL94] G Itkis and L Levin. Fast and lean self-stabilizing asynchronous protocols. In *FOCS94 Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 226–239, 1994.

[KP93] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[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.

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

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

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

[Sch93] M Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Trans. on Info. Theory*, IT-29(1):23–35, January 1983.

[Spi88] John M. Spinelli. Reliable communication. PhD thesis, MIT, Lab. for Information and Decision Systems, December 1988.

[Tan93] A. Tannenbaum. *Computer Networks*. Prentice Hall, 1993.

[Tel89] Gerhard Tel. *The Structure of Distributed Algorithms*. PhD thesis, University of Utrecht, also published by Camridge University Press, 1989.

[Var93] G Varghese. Self-stabilization by local checking and correction (PhD thesis). Technical Report MIT/LCS/TR-583, MIT, 1993.

[Var94] G Varghese. Self-stabilization by counter flushing. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

[Var98] George Varghese. Self-stabilization by counter flushing. "http://dworkin.wustl.edu/varghese/PAPERS", August 1998.