Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-94-14

1994-01-01

Self-stabilization by Counter Flushing

George Varghese

A useful way to design simple and robust protocols is to make them self-stabilitizing. We describe a simple technique for self-stabilization called counter flushign which is applicable to a number of distributed algorithms. A randomized version of counter flushing is shown to have extremely small expected stabilization time. We show how our technique helps to crisply understand and improve some previous distributed algorithms. Then we apply it to a variety of total algorithms for deadlock detection, propagation of information with feedback, resets and snapshots. Our stabilizing snapshot protocol has much better complexity than the previous stabilizing non-blocking snapshot protocol.... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Varghese, George, "Self-stabilization by Counter Flushing" Report Number: WUCS-94-14 (1994). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/336

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/336

Self-stabilization by Counter Flushing

George Varghese

Complete Abstract:

A useful way to design simple and robust protocols is to make them self-stabilitizing. We describe a simple technique for self-stabilization called counter flushign which is applicable to a number of distributed algorithms. A randomized version of counter flushing is shown to have extremely small expected stabilization time. We show how our technique helps to crisply understand and improve some previous distributed algorithms. Then we apply it to a variety of total algorithms for deadlock detection, propagation of information with feedback, resets and snapshots. Our stabilizing snapshot protocol has much better complexity than the previous stabilizing non-blocking snapshot protocol. Hence it can be used to improve the complexity of general compilers that convert arbitrary asynchronous protocols into stabilizing equivalents.

Self-stabilization by Counter Flushing

George Varghese

WUCS-94-14

December 1994

A useful way to design simple and robust protocols is to make them self-stabilizing. We describe a simple technique for self-stabilization called counter flushing which is applicable to a number of distributed algorithms. A randomized version of counter flushing is shown to have extremely small expected stabiization time. We show how our technique helps to crisply understand and improve some previous distributed algorithms. Then we apply it to a variety of total algorithms for deadlock detection, propagation of information with feedback, resets and snapshots. Our stabilizing snapshot protocol has much better complexity than the previous stabilizing non-blocking snapshot protocol. Hence it can be used to improve the complexity of general compilers that convert arbirary asynchronous protocols into stabilizing equivalents.

Computer and Communications Research Center
Washington University
Campus Box 1115
One Brookings Drive
St. Louis, MO 63130-4899

Self-stabilization by Counter Flushing

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

January 17, 1995

Abstract

A useful way to design simple and robust protocols is to make them self-stabilizing. We describe a simple technique for self-stabilization called *counter flushing* which is applicable to a number of distributed algorithms. A randomized version of counter flushing is shown to have extremely small expected stabiization time. We show how our technique helps to crisply understand and improve some previous distributed algorithms. Then we apply it to a variety of *total algorithms* for deadlock detection, propagation of information with feedback, resets and snapshots. Our stabilizing snapshot protocol has much better complexity than the previous stabilizing non-blocking snapshot protocol. Hence it can be used to improve the complexity of general compilers that convert arbirary asynchronous protocols into stabilizing equivalents.

1 Introduction

Informally, a protocol is *self-stabilizing* if when started from an arbitrary global state it exhibits "correct" behavior after finite time. While typical protocols are designed to cope with a specified set of failure modes (e.g., message loss, link failures), a self-stabilizing protocol essentially copes with a set of failures that subsumes most previous categories and is also robust against transient errors (such as memory corruption and malfunctioning devices that send out incorrect messages). There is evidence from real networks that such transient errors do occur (e.g., the ARPANET [Ros81, Per83]) and cause systems to fail unpredictably. Thus stabilizing protocols are attractive because they offer *increased robustness* (especially to transient faults) as well as *potential simplicity* (because a stabilizing protocol can avoid the need for a slew of independent mechanisms to deal with a catalog of anticipated faults.)

Self-stabilizing protocols were introduced by Dijkstra [Dij74]. Since then, they have been studied by various researchers (e.g., [BP89, GM90, KP90, DIM90, IJ90a], [IJ90b, AG90, AKY90]). However, only recently has there been a study of general techniques for self-stabilization. A landmark paper is that due to Katz and Perry [KP90], which showed how to compile an arbitary asynchronous protocol into a stabilizing equivalent. The basic idea is to add a leader node periodically do a snapshot of the network and reset the network if a global inconsistency is detected. We call this idea *global checking and correction.* However, the general transformation is expensive and so the search has continued for techniques that are less general and more efficient than those of Katz and Perry but still apply to a number of useful protocols.

A paper by [AKY90] suggests how to make a spanning tree protocol stabilizing by having nodes detect inconsistent *global* states by checking the states of neighbors. In [APV91, Var92] this notion is formalized using the notion of *local checkability* – protocols that are locally checkable can be checked for global inconsistency by using more efficient local checking. [APV91, Var92] goes further and defines the class of *locally correctable* protocols that can be corrected into good global states by local correction actions.[1] In [APV91, Var92] local checking and correction is used to design stabilizing protocols for mutual exclusion, the end-to-end problem, and network reset.

However, not every protocol is locally correctable. Another general method, suggested in [Var92], is called *local checking and global correction* by which any locally checkable protocol can be stabilized using a stabilizing network reset protocol. Using the optimal reset protocol described in [AKM$^+$93], this leads to protocols that stabilize in time proportional to the network diameter.

This paper describes another general technique, called *counter flushing* that is applicable to some protocols that are *neither* locally checkable or correctable. The setting is that of a leader which wishes to periodically deliver a message to every network node (and sometimes to every link) in the network. By attaching a simple counter to the state of every node and to every message, and by using a few simple checks, we can ensure that the protocol will begin to work correctly regardless of the initial messages and node states. In particular we advocate a randomized version of counter flushing (in which each new message is numbered with a random counter value chosen from a sufficiently large space) that has extremely fast expected stabiization time.

The method appears to be applicable to several *total algorithms*, ([Tel89]) which are essentially algorithms that involve the cooperation of all nodes in the network. Some protocols

[1] that affect the state of a pair of neighbors.

to which this technique is applicable include token passing [Dij74], propagation of information with feedback [Seg83], deadlock detection [Mis83], network resets [AG90], and non-blocking network snapshots [CL85]. Our stabilizing version of the Chandy-Lamport snapshot protocol improves the complexity of the Katz-Perry version and hence can be used as an improved component of the general transformation described in [KP90].

In some cases, the solutions provided by counter flushing can also be provided by local checking and correction. However, the method of local checking requires a fairly tedious enumeration of the protocol invariants which need to be checked; the addition of local checking also has a fair amount of complexity [Var92]. Also, taking correct snapshots of local state requires some careful synchronization which makes actual implementations ([CSV89]) somewhat tricky. By contrast, the modifications required by counter flushing are extremely simple. Thus we believe that counter flushing is to be preferred in practice even when both methods are applicable.

Besides providing new results, a useful paradigm should be able to unify and help understand previous work in the field. For example, we show that Dijkstra's N-state example [Dij74] can be understood very simply using counter flushing; in fact the requirement for $O(N)$ states follows almost immediately from the general paradigm.[2] We even show that this protocol can be easily extended to a message passing version which appears to be simpler than the token passing protocols used in today's Local Area Networks. The counter flushing paradigm also exposes a basic unity behind Dijkstra's token passing protocol, results on stabilizing Data Links [AB89, GM90], and results on stabilizing request-response protocols [Var92].

The rest of the paper is organized as follows.

First in Section 2 we describe our model of computation and then describe our assumptions about links and network topologies. Next, in Section 3, we introduce the counter flushing paradigm by describing a message-passing version of Dijkstra's token ring protocol. In this section, we also introduce the simple idea of randomized counter flushing and compare our solution with existing work in stabilizing Data Links, especially the elegant solution of [AB89]. In Section 5.1, we extend the use of counter flushing on a ring to provide stabilizing deadlock detection by transforming a protocol due to [Mis83]. In Section 6, we extend counter flushing to trees as exemplified by the well-known Propagation of Information with Feedback (PIF) protocol due to Segall. In Section 7, we describe how to use counter flushing to produce a stabilizing reset for a general network. This reset protocol in turn can be used to stabilize certain diffusing computations, as exemplified by a stabilizing version of the Chandy-Lamport

[2]We have previously shown that Dijkstra's second example can be formally derived using the ideas of local checking and correction [AGV92].

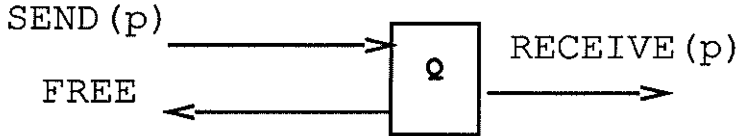


Figure 1: A Unit Capacity Data Link

protocol that stabilizes in time proportional to the network diameter. Then we briefly conjecture that counter flushing techniques are applicable to virtual circuit problems as well. Finally, in Section 10 we present our conclusions.

2 Models

2.1 Modelling Tool – I/O Automata

In the IOA model, transitions by which the environment affects the automaton (e.g., SEND) are called *Input* actions while transitions by which the automaton affects the environment (e.g., RECEIVE) are called *Output* actions. Input actions are under the control of the environment while output actions are under the control of the automaton. Finally, there are *internal actions* which only change the state of the automaton without affecting the environment.

Formally, an I/O Automaton (henceforth IOA) is characterized by its *state set* S, a *action set* A, an action signature G (that classifies the action set into input, output, and internal actions), a *transition relation* $R \subseteq S \times A \times S$, a set of initial states $I \subseteq S$. The set of output and internal actions are called the *locally controlled* actions of the automaton. Fairness is specified by dividing the set of locally controlled actions into a finite number of equivalence classes. For stabilization, we will often limit ourselves to a special type of IOA that we call a *UIOA* (for *uninitialized IOA*) in which the state set is finite and $I = S$. In other words, any state is a possible start state for a UIOA.

An action a is said to be *enabled* in state s if there exist $s' \in S$ such that $(s, a, s') \in R$. By definition, input actions are always enabled. (This is quite natural for many message passing systems, where messages can arrive at any time). When the automaton "runs" it produces an execution. An *execution fragment* of the automaton is modeled by 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 is an execution fragment that begins with a start state and is fair. An execution fragment E is fair if every locally controlled class C is given a "fair turn"; more formally, if some action of C is enabled in some state s of E, then either some action in E occurs after s or there is some later

state in which no action of C is enabled.[3]

There is a notion of composition of automata that allows automata to be "plugged together" using simultaneous performance of shared actions. For example, consider a node automaton that had an action to send a packet SEND(p) as an output action. Since this is the same name as the input action for the channel automaton shown in Figure 1, when the two automata are composed, whenever the node performs a SEND(p) output action, the channel simultaneously performs a SEND(p) input action. The formal details can be found in [LT89]; intuitively the composition of automata is a new automaton whose state set is the cross-product of the component state sets, whose transition relation is obtained from the component automata in the natural way, and whose locally controlled classes are the union of the locally controlled classes of the component automata.

Finally, a behavior is the subsequence of an execution consisting of external (i.e., input and output) actions. Thus each automaton generates a set of behaviors. We specify the correctness of a protocol using a set of behaviors P; an automaton A is said to solve P if the behaviors of A are a subset of P. This definition reflects a belief that the correctness of an automaton should be specified in terms of its externally observable behavior. For example, to specify a FIFO Data Link we might require that the sequence of received packets be identical to the sequence of sent packets.

2.2 Modelling Bounded Links using Initially Bounded Asynchronous Models

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

Unfortunately, this is no longer true in a stabilizing setting. *If the link can store an unbounded number of packets, it can have an unbounded number of "bad" packets in the initial state.* It has been shown [DIM91] that almost any non-trivial task is impossible [4] in such a setting. Thus the original simplification of considering only unbounded links is no longer valid. Since real links are bounded and (as we show below) bounded links can be modeled elegantly, we restrict ourselves to bounded link models.

[3] One can think of the automaton running under the control of a scheduler that samples each class to see if it is enabled and if so, giving some enabled action in the class a turn. Of course, if no action in the class is enabled whenever the scheduler samples the class, the scheduler is under no obligation to give this class a turn.

[4] More precisely it is impossible to provide bounded stabilization time

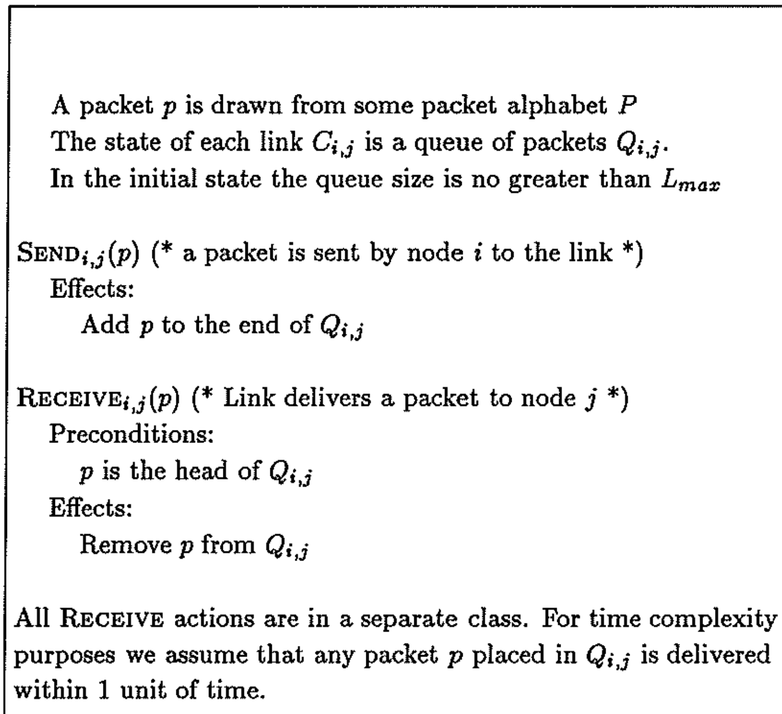
In previous work, we have modelled bounded links as a unit capacity Data Link or UDL for short. Intuitively, a UDL can store at most one packet at any instant. We can show [Var92] that a UDL can be implemented over real physical channels and can easily be generalized to bounded capacity data links. Roughly, [Var92] a UDL can be thought of as a model of a reliable Data Link protocol that only delivers one message at a time (i.e., it uses a window size of 1), A UDL can be implemented (see [Var92]) by an underlying stabilizing Data Link that sends and receives acknowledgements.

However, many real protocols, especially those that work over very reliable (e.g., fibre) links *do not use an underlying Data Link protocol.* Consider a single link of a token passing ring like the FDDI or IBM token ring. It is a single piece of fibre of bounded length (say up to 1 mile) that connects two nodes. There is no data link protocol between the two nodes; messages are simply relayed between the two nodes on the assumption that the links are mostly reliable. However, the number of messages stored on a link is always bounded. For a mile long link, assuming speed of light limitations, we have a 5 usec propagation delay. Suppose the nodes transmit 20 byte messages (i.e., tokens) at 100 Mbit/sec. Then roughly 3 tokens can be stored at any instant in the ring. Of course, the reason why the "queue" that models the stored messages on a link is always bounded is that the sender and receiver are *transmitting synchronously.* The receiver is taking token messages out as fast as the sender can input these messages.

Modelling the synchrony between transmitter and receiver is possible but is somewhat involved and also tends to imply that our basic idea is confined to such synchronous systems. Instead, we propose the following model of a bounded link. We model a bounded link (as usual) as a queue such that packet send events add elements asynchronously to the head of the queue and packet receive events remove elements asynchronously from the tail of the queue. The only twist is that:

- For self-stabilization we assume that in the initial state all *all links queues are bounded.* For example, in the token passing example, the bound was 3. However, *we do allow the queue to grow unboundedly after that.*
- For time complexity purposes, we assume that any message stored in a link queue will be delivered 1 unit of time later, regardless of the size of the queue. For example, the token passing example, 1 unit of time would correspond to 5 usec.

At first glance, this seems like "cheating". The second assumption seems absurd because we are guaranteeing a fixed delay for every message regardless of whether there is 1 or a million messages ahead of it. However, we have seen that in the real system the queue is always



A packet p is drawn from some packet alphabet P
The state of each link $C_{i,j}$ is a queue of packets $Q_{i,j}$.
In the initial state the queue size is no greater than L_{max}

SEND$_{i,j}(p)$ (* a packet is sent by node i to the link *)
Effects:
Add p to the end of $Q_{i,j}$

RECEIVE$_{i,j}(p)$ (* Link delivers a packet to node j *)
Preconditions:
p is the head of $Q_{i,j}$
Effects:
Remove p from $Q_{i,j}$

All RECEIVE actions are in a separate class. For time complexity purposes we assume that any packet p placed in $Q_{i,j}$ is delivered within 1 unit of time.

Figure 2: Formal Model of an Initially Bounded Data Link

bounded and so the time delay assumption holds. In fact, the only reason we are allowing the queue to grow unboundedly in our model is to take advantage of the simple modelling machinery that exists for asychronous protocols (i.e. I/O Automata) without explicitly modelling time except for time complexity. However, we claim the following fact informally: if a given protocol P can be shown to be stabilizing in our initial bounded but asynchronous model, then P is stabilizing is in the bounded, synchronous model (which we claim is the "real-life" model). Intuitively, this is because the initially bounded, asynchronous model has a richer set of behaviors than the bounded synchronous model.[5]

A formal specification of an initially bounded Data Link is shown in Figure 2

[5]Like all seemingly "obvious" statements a precise statement and proof would be worthwhile. Thus it is more accurate to say that we take this as our article of faith.

2.3 Network Topologies

In what follows we consider various topologies consisting of nodes connected by initially bounded Data Links. Except for the case when we consider a ring topology (Section 3), we assume that links are symmetric – i.e., between any pair of neighbors i and j there is exactly one link in either direction.

To apply the counter flushing paradigm, we also need the assumption that there is a leader node in the network, There are many stabilizing protocols to construct a leader, especially the protocol of [AKM$^+$93] that calculates this leader in $O(D)$ time ignoring polylog factors. We assume therefore that a fixed node is designated as the leader. For the case of a general network, we also assume that there is also a pre-computed spanning tree of the network rooted at the leader. Once again this is not a restrictive assumption: given a unique leader a spanning tree rooted at the leader can easily be calculated in $O(D)$ time as shown in [DIM89].

3 Counter Flushing on a Token Ring

The protocol described in Figure 3 is a simple token passing ring and is a message passing version of the first example in [Dij74]. Dijkstra's version was described in a simple shared memory model. We have chosen to use a message passing model uniformly for describing the counter flushing paradigm. The message passing model also introduces some subtleties not present in Dijkstra's original protocol – for instance, the size of the counter now has to be increased to take into account messages that can be stored on the channel.

We will use our initially bounded Data Link model. The nodes in the ring are numbered from 0 to $n-1$. All arithmetic on node indices is assumed to be mode n, Thus between any nodes i and $i+1$ we assume there is a link $C_{i,i+1}$ which was specified in Figure 2. Assume that the nodes are laid out in a ring with node 0 at the top and the indices increasing in clockwise order.

The protocol is very simple. A token messages carries a counter and the state of each node also consists of a counter; a counter is simply an integer in the range 0 to *Max.* Each node periodically retransmits its counter value in a token message. Node 0 is a special process[6] whose local protocol is different from the other nodes. When Node 0 receives a token from its clockwise neighbor, if the counter in the token *is equal* to Node 0's local counter, then Node 0 changes its counter value using a function CHOOSE which returns an arbitrary counter value

[6]Equivalent to the Ring Monitor in the IBM Token Ring

A token message 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 $0..Max$
Assume there are n nodes numbered from 0 to $n-1$.
All addition and subtraction of process indices is mod n.

$\textsc{Choose}(Max, c)$
Function which non-deterministically returns any integer not equal to
c in the range from $0..Max$. Later we will discuss specific implementations of this function.

$\textsc{Receive}_{n-1,0}(Token, c)$ (* Node 0 receives token from Node $n-1$ *)
Effects:
If $c = count_0$ then (* token counter matches node counter *)
$count_0 = \textsc{Choose}(Max, c)$ (*pick any value other than c *)

$\textsc{Receive}_{i-1,i}(Token, c), i \neq 0$ (* 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 message*)

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

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

Figure 3: Code for node processes in a token ring

that is different from Node 0's stored value. Later we will describe three specific realizations of the CHOOSE function that guarantee self-stabilization.

When any node i other than 0 receives a token from its clockwise neighbor $i-1$, Node i does the following. If the counter value in the token (say c) *is different* from the counter stored at node i, then node i changes its stored counter value to c.

Its important to understand the behaviors of this protocol when it is in a good state. So consider the state in which all token messages on links have a counter value c, and the counter values at all nodes except 0 is also c. The counter value at node 0 is $c' \neq c$. In that case we say that node 0 has the token. Eventually node 0 will transmit a token message containing c'. In that case we say that the token has left Node 0 and is in the link from Node 0 to Node 1. When this message reaches Node 1, Node 1 sets its counter value to c'. Now we say that Node 1 has the token. This process continues with the token moving clockwise until Node $n-1$ receives has the token and transmits it to Node 0; Node 0 then chooses a new value and the cycle continues.

Thus in good states the ring can be partitioned into two bands. The first band starts with the leader and continues up to (but not including) the first counter value (either in a token message or at a node) whose counter value is different from that of Node 0. The remainder of the ring (including links and nodes) is a second band containing counter values that are different from Node 0. As the token rotates round the ring, the first band gets larger until it spans the entire ring; then Node 0 chooses a new counter value and the first band shrinks to only containing Node 0; and so the cycle continues.

3.1 Stabilization for Token Ring

Consider the token passing protocol defined by the composition of the node automata of Figure 3 and the link automata defined in Figure 2. Define a *counter change step* as a $\text{RECEIVE}_{n-1,0}(token, c)$ event with $c = count_0$. We define a *ring rotation time* equal to $2N$ time units (i.e., the time it takes for a message to travel around the ring with a unit delay at each node and link.)

We have the following lemma:

Lemma 3.1 *A counter change step will occur in 1 ring rotation time starting from any state.*

Proof: (Sketch) A counter change step occurs when the leader receives a counter equal to its own stored value. If this does not happen in 1 ring rotation time, then Node 0's value remains fixed in this interval. Thus Node 0's value will travel all the way around the ring in tis interval, causing a counter change step. □

Define a *fresh counter change* step as a counter change step which results in a state in which i) $count_0 \neq count_i, i \neq 0$ and ii) for any $(token, c)$ message present in any link, $count_0 \neq c$. (Intuitively, this is an event which causes Node 0 to pick a counter value that is not equal to any counter values stored in other nodes or links.) We will sometimes also say that after a fresh counter change step the leader (i.e., Node 0) has picked a fresh counter value. We have the following lemma:

Lemma 3.2 *In 1 ring rotation time after a fresh counter change step, the token passing protocol will reach a good state.*

Proof: (Sketch) After a fresh counter change event, Node 0 has a value c that is not present anywhere else on the ring. Now Node 0 will not do another counter change step until it receives a token message with counter value c; it is easy to see that this will occur only after the value c travels all the way around the ring[7]. Thus immediately before the next counter change step, all nodes have stored counter value c and any token messages on links have counter c, which is a good state. □

Thus to prove that the token passing protocol stabilizes, all we have to do is to prove that it does a fresh counter change step in bounded time. In order to show this we consider 3 implementations of the CHOICE function, the *Increment, Random, and Random-Increment* functions.

- The Increment function $\text{INCREMENT}(Max, c) = c + 1 \bmod c$.
- The Random function $\text{RANDOM}(Max, c)$ chooses a random value other than c in the range $0 \ldots Max$.
- The Random-Increment is a composite of these two schemes: it assumes the counter c contains $2m$ bits, and so we can regard c as the concatenation of two m bit strings d and e. Then we apply the Increment function to d and the Random function to e and return the concatenation of the two resulting strings.

[7]More precisely, the fresh counter change step causally precedes the next counter change step.

We can now prove results about the stabilization time using these three implementations of the CHOICE function. The easiest is to show that the Random function provides fast stabilization time with high probability.

The stabilization time depends heavily on the maximum number of distinct counter values that can be present in the initial state. We denote this quantity by c_{max}. It is easy to see that $c_{max} = n(1 + L_{max})$, since each link can store L_{max} values in the initial state and each node can store 1 value.

Theorem 3.3 *If the token passing protocol uses the Random function as its* CHOICE *function and $Max > c_{max}$, then with probability $(Max - c_{max})/Max$, the stabilization time is 2 token rotation times.*

Proof: We know from Lemma 3.1 than a counter step event occurs in 1 ring rotation time. Since the leader is picking a random value, the leader picks a *fresh counter value* with probability $(Max - c_{max})/Max$. But by Lemma 3.2 in 1 more ring rotation time, the protocol is in a good state. □

In practice, it is not hard to make the stabilization time of this protocol 2 ring rotation times with very high probability. For example, with a 1000 node ring transmitting at 100 Mbp/s and assuming links that are at most ten miles long between nodes, we have $c_{max} = 31,000$. (It is unlikely that a real token ring exists with such pessimistic parameters). If c is a 32 bit counter, then the stabilization time is 2 ring rotation times with probability $(1-2^{-17}/1)$. With a 16-bit counter the value is the probability is 0.5.

It is tempting to conclude that the expected stabilization time of the protocol is simply $(Max/(Max-c)$ ring rotation times. Unfortunately, this is not true in our model in which links can have unbounded storage. This is because after each application of the choice function we potentially increase the number of counters stored in the network by 1. Thus after $Max - c_{max}$ iterations the number of distinct counters stored in the network can be equal to Max and so the probability of obtaining a fresh value can drop to zero (which results in an infinite expectation)!

There are two ways to deal with this. The first is to settle for the high probability result in the unbounded model; note that in the real world setting the number of counters in each state is always bounded and the expected value is indeed $(Max/(Max - c)$. The other way to deal with the problem is to use slightly more sophisticated implementations of the CHOICE function.

Theorem 3.4 *If the token passing protocol uses the Increment function as its* CHOICE *function and $Max > c_{max}$, then the protocol stabilizes in c_{max} ring rotation times.*

Proof: (Sketch) Suppose the initial value of $count_0$ is c. Since $Max > c_{max}$, there is some value c' such that $c' - c < c_{max}$ and such that in the initial state no node or token message has counter value c. But we know from Lemma 3.1 that node 0's counter will increment within 1 ring rotation time. Thus within $c_{max} - 1$ ring rotation times, Node 0 will do a counter change step that results in $count_0 = c'$. We claim that this is a fresh counter change step. This is because the value c' was not present in the initial state; in the interval till this counter change step, the value c' cannot be added to any node and any link. This is because only Node 0 produces new counter values (see Figure 3), and Node 0 during this interval has only produced values in the range $[c, c' - 1]$. □

It is easy to see that the Random-Increment choice function can provide the best features of both schemes: it can have fast expected stabilization time (of close to 2 ring rotation times) and also a larger deterministic bound (c_{max} ring rotation times). Note that the expectation of this function is well-behaved (unlike the RANDOM function) because when we calculate the expectation by summing a series, there are only $c_{max} + 1$ terms. The analysis is a composite of the two analyses shown above.

In the sequel we will assume that the CHOICE function is the Random-Increment scheme, where the value of *Max* is chosen so that the expected time is $O(R)$ where R is the worst case delay from the leader to any other node.[8]

4 Counter Flushing Paradigm

Suppose in a network 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 [Var92], for example, we implement local snapshots and resets between a pair of nodes using such a request-response protocol initiated by the leader of each link subsystem. In order to properly match responses to requests, the sender numbers each request with a counter. Let m be the number of packets that can be in transit between the sender and responder and let n be the number of responders. Then the sender uses a counter that has $Max > m + n + 1$ distinct values. For example, in [Var92] we used a counter in the range 0...3 because there can be at most two packets in transit in a link subsystem and there is only one responder.

Responders only accept *Request* packets with a number different from the last *Request*

[8]Rather surpisingly, in some cases, the simple INCREMENT function also has $O(R)$ deterministic stabilization time! However, the proofs of these results are somewhat harder and are deferred to the final paper. Thus we prefer the Random-Increment scheme because of its generality and simplicity.

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 chooses a new counter value (using one of the three CHOICE function implementations we just described) and starts a new phase of sending *Request.*

The size of *Max* and the CHOICE function ensure that within bounded time[9], the sender will reach what we call a "fresh" counter value – i.e. a counter value that is not currently stored in either the links or the responders. We call the method counter flushing because the request-response protocol must guarantee the following "flushing" property. *Suppose the sender sends a request numbered c, where c is a fresh value. Then after all matching responses to this request arrive, there must no counter values other than c that are stored in the links or at the responders.* In other words, the sending of a freshly numbered request and the receipt of all matching responses, should "flush" the links and responders of "old" counter values.

Thus the token passing protocol described here and the one due to [Dij74] can be simply understood as counter flushing in a unidirectional ring. The flushing is guaranteed because in a ring, the leader is connected to itself by a sequence of unidirectional links. This is also true for a pair of neighbors connected by a pair of unidirectional links (essentially this a two node ring) and thus applies to request-response protocols [Var92], Data Link protocols [AB89] and token passing between a pair of nodes [DIM91]) as long as links are initially bounded.

It is interesting to compare our paradigm with the elegant result of Afek and Brown [AB89] to build a stabilizing Data Link protocol. Their major result applies to initially unbounded links where they suggest using bounded length counters of size greater than 2 but such that the sequence of counter values used is aperiodic. A trivial corollary of their results 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 messages. They even suggest the use of randomization (i.e., 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. With a little effort, it is possible to see that Dijkstra's token ring protocol can be considered to be a Data Link protocol in the Afek-Brown sense between the ring leader and any arbitrary node on the ring. However, the paradigm does not extend to general networks as ours does 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.

[9]in case of the Random-Increment choice function the expected time is bounded

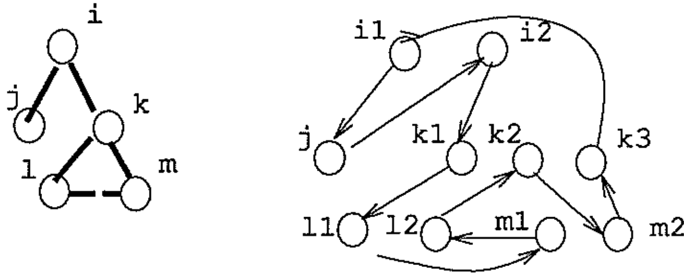


Figure 4: Creating a Virtual Ring that Spans all node

For the rest of this paper, we discuss the application of the Counter Flushing paradigm to other distributed algorithms and to networks other than rings.

5 Embedding Rings in General Networks

We have already seen that a *link subsystem*, consisting of two nodes connected by two unidirectional links, is a special case of a ring and many of the stabilization results for link subsystems follow from the stabilization results for a ring. We now consider embeddings of a ring in a general network, especially as applied to the problem of distributed deadlock detection.

First, note that once we have a leader and a spanning tree it is easy to have a stabilizing distributed algorithm that constructs "virtual rings" that span all nodes and even all links. A virtual ring that spans all nodes is an Euler tour that starts and ends at the leader and in which each node is visited at least once. A virtual ring that spans all links is a cyclic path that starts and ends at the leader and in which each link is visited at least once. Since we have assumed that our network consists of symmetrical links (i.e., for every pair of neighbors i and j, there is a link from i to j and one from j to i), the network has an Euler Tour.

For instance, to create a virtual ring that spans all links, each non-tree link is assigned to one of its two end-points arbitrarily. Each node is then broken up into k nodes where k is the degree of the node; then the resulting set of nodes is connected by a virtual ring essentially formed by an in-order traversal of the tree; non tree-links are, however, visited for the first time from the assigned node. Figure 4 shows how to construct an Euler tour for the graph shown in the left in which there is only 1 non-tree edge between l and m. The figure on the right shows the resulting virtual ring. It is easy to see that we visit every link exactly once in each direction, and so the length of the tour is twice the number of links in the original graph.

It is easy to see that we can do token passing on an arbitary network using the counter

flushing paradigm of the previous section. However, a more interesting example is deadlock detection. We adapt an algorithm due to Misra and make it stabilizing.

5.1 Misra's deadlock detection algorithm

Suppose we have some message driven distributed algorithm that is executing in a network. When a node gets a message it does some local computation and possibly sends out more messages. We say that the algorithm is *deadlocked* if there is no protocol message in transit between nodes.[10] We assume that deadlock is a *stable property* – i.e., once it is true it remains true unless corrected for. An important application is that of Pessimistic Distributed Simulation [CM81] in which deadlock, if it occurs, must be detected and corrected for.

In Misra's scheme to detect deadlock [Mis83], a special message called a marker periodically circulates through the network. Assume that each tour of the marker starts from a designated leader node[11] and visits every link once[12] before returning to the leader. Each node i keeps a flag $busy_i$ which is set to *true* whenever a message of the underlying protocol arrives at the node. A marker message carries a flag b which is set to *false* at the start of the tour by the leader. When a node i receives a marker (note that node i may receive the marker several times), node i sets the marker flag to *true* if $busy_i = true$ and sets $busy_i = false$. Finally, the leader declares deadlock if at the end of the tour the marker flag is *true*.

This algorithm works correctly if all links are assumed to be FIFO. Suppose there is a message in transit between nodes u and v at the start of a marker tour. Then since the marker must visit link (i, j) and the link is FIFO, the message must be delivered to j before the marker visits j for the last time in this tour. Consider the first time that the marker visits j after the message is delivered; then the marker will find that $busy_j = true$ which will result in setting the marker flag. Thus at the end of the tour, the leader will not detect deadlock. Conversely suppose the system is deadlocked at the start of a marker tour. Then by end of the tour, all nodes will have set $busy_i = false$ since no messages will will be delivered to nodes during the tour. Then on the next tour it is easy to see that the marker flag will never be set and the leader will declare deadlock. Thus deadlock detection occurs at most 3 tours after deadlock actually occurs.

---

[10] In Misra's presentation, the system is deadlocked when no message is in transit and no node is doing local computation. For ease of presentation, we assume local computation is instantaneous. It is easy to modify the stabilizing algorithms to take into account local computation.

[11] Misra's scheme does not use a leader; since we need a leader anyway to implement counter flushing we prefer to describe it using a leader

[12] if links are bidirectional, every link is visited once in each direction

A stabilizing deadlock detection algorithm may incorrectly declare deadlock for some initial period. However, after some bounded period of time, we wish the algorithm to declare deadlock if and only if the network is really deadlocked. Despite the possibility of initial errors such an algorithm may still be useful because of its robustness and if the initial "false alarms" only cause harmless diagnostic procedures to be run.

To make Misra's scheme fault-tolerant in this sense, we assume that there is a virtual ring embedded in the network that spans all network links (see Section 5). Recall however, that the virtual ring (see Figure 4) consists of virtual nodes and that each physical node may simulate multiple virtual nodes. As usual we number any one of the virtual nodes belonging to the leader as node 0 and number virtual nodes in the ring with consecutive indices. The code for each virtual node is shown in Figure 5. We augment Misra's scheme by making having a marker be a special kind of token message which carries a flag b as well the usual counter c.

To preserve the counter flushing paradigm each virtual node has to periodically retransmit its counter value to its successor. (This is necessary to ensure that the leader periodically changes its counter value as in Lemma 3.1). However, in Misra's scheme when a virtual node transmits a marker it updates the marker flag and clears its own flag. Thus to simulate Misra's algorithm we wish a node to clear its own flag only on its first transmission after it receives a new marker and not on subsequent retransmissions. Thus each virtual node i has an additional flag $token_i$ which is set to *true* on receipt of a new marker (i.e., a marker with a different counter value than the counter value stored at i). While virtual node i periodically retransmits its counter value to its successor, virtual node i will only clear $busy_i$ when $token_i$ is *true*, after which it sets $token_i$ to *false*.

We briefly sketch the proof of stabilization of this modified version of Misra's deadlock detection algorithm. We use the same terminology as in Section 3 except that we use virtual nodes in place of physical nodes. Refer to Section 3 for the definitions of counter change and fresh counter change events.

Recall, too that we said that the token ring was in a good state if the sequence of counters values around the ring (including counters at nodes and counters in token messages in transit) has at most one change of value. If the change of value first occurs at node i, we said that node i has the token. We define the deadlock detection scheme to be in a good state iff i) the sequence of counter values around the virtual ring has at most one change of value and ii) $token_i = true$ at a virtual node i iff the first change in counter values occurs at node i.

It is easy to prove a version of Lemma 3.1 for the stabilizing deadlock detection scheme which shows that the leader will change its counter value within 1 rotation time around the virtual ring. Once again this follows because each node periodically retransmits its counter

A token message is encoded as a tuple $(Token, c, b)$ where c is an integer in the range $0..Max$ and b is a Boolean Flag
The state of each virtual node i consists of an integer $count_i$ in the range $0..Max$ and two boolean flags $token_i$ and $busy_i$.
We assume that each virtual node is numbered from 0 to $n-1$.
All addition and subtraction of process indices is mod n.

RECEIVE$_{n-1,0}(Token, c, b)$ (* Virtual Node 0 receives token from Virtual Node $n-1$ *)
Effects:
If $c = count_0$ then (* token counter matches node counter *)
If $busy_0 = false$ and $b = false$ then (* marker flag set or node busy?*)
DECLARE_DEADLOCK
$busy_0 = false$ (* reset marker for next tour *)
$token_0 = true$ (* give node an opportunity to transmit and clear busy flag *)
$count_0 = \text{CHOOSE}(Max, c)$ (* choose a new counter value *)

RECEIVE$_{i-1,i}(Token, c, b), i \neq 0$ (* Virtual Node i receives token from Virtual Node $i-1$ *)
Effects:
If $c \neq count_0$ then (* token counter differs from node counter *)
$count_i = c$ (*set value to counter in token message*)
$token_i = true$ (* give node an opportunity to transmit and clear busy flag *)
If $b = true$ then $busy_i = true$

SEND$_{i,i+1}(Token, c, b)$, (* Virtual Node i sends token to Virtual Node Node $i+1$ *)
Preconditions:
$c = count_i$ (* counter of token matches node counter *)
$b = busy_i$
Effects:
If $token_i = true$ then (* clear busy flag on transmission only if indicated by token flag *)
$busy_i = token_i = false$;

RECEIVE$_{u,v}(Data)$, (* Physical Node v receives a Data Message *)
Effects:
For all virtual nodes i that belong to node v
$busy_i = true$

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

Figure 5: Code for Deadlock Detection using a Virtual Ring of Processes

value to its successor. A version of Lemma 3.2 also holds – i.e., in 1 virtual ring rotation time after a fresh counter change step, the deadlock detection protocol will reach a good state.

As before, after a fresh counter change event, Node 0 has a value c that is not present anywhere else on the ring. Now Node 0 will not do another counter change step until it receives a token message with counter value c; it is easy to see that this will occur only after the value c travels all the way around the ring. As before, this ensures that before the next counter change step, all nodes have stored counter value c and any token messages on links have counter c. But is also ensures that every node i has $token_i = false$ which ensures the protocol is in a good state.

Thus by the counter flushing pardigm, using any of the three CHOOSE functions we guarantee that starting from an arbitrary state, within bounded time (in either an expected or worst case sense), the deadlock detection protocol is in a good state. For instance, the random function guarantees that the protocol will stabilize to a good state after expected time $4m$, where m is the number of network links.

Any complete tours after this point will detect deadlock correctly. The proof of correctness is in fact identical to Misra's proof except that we also need to argue that a virtual node does not clear its *busy* flag prematurely.

One problem with Misra's protocol is that takes $O(m)$ where m is the number of network links while other deadlock detection protocols take $O(n)$, where n is the number of nodes. An easy way to make Misra's protocol faster is to use multiple markers; each marker is responsible for traversing some subset of the links, but each marker tour takes $O(D)$ where D is the network diameter. The leader assumes a phase of the algorithm is completed when all markers have completed their tours.

Other deadlock detection protocols, (for example one due to Chandy) avoid traversing every link by having the sending and receiving ends of each link keep a counter of messages sent and received respectively. After visiting each node, discrepancies in the two counters associated with a link can be used to infer the presence of transit messages without the "flushing" mechanism used in Misra's scheme.

Unfortunately, there are two problems with this scheme in a practical stabilizing setting – first the relation between the send and receive counters of a link can be arbitrary in the initial state, and second the counters must be finite and have to allow wrap-around. The first problem can be stabilized (assuming FIFO links) by having each node i periodically send its send counter for the link to each neighbor j; when j receives the message, j sets its receive counter for the link to i equal to the counter sent by i. For the second problem, we note that often all that is needed by the deadlock detection protocol is two conditions: i) if at any instant

there is a message in transit from i to j, then the send counter at i should not be equal to the receive counter at j ii) If at some instant t_1, the send counter of i is not equal to the receive counter at j, then the send counter of i at some later instant t_2 should not be equal to the receive counter of j at t_1.

Both conditions are trivially met if the send and receive counters of all links are synchronized initially and the counters are integers (since in this case the send counter is always non-decreasing.) However, to make these conditions work with finite sized counters, we need to assume that the counters are large enough such that i) that the counter size is larger than the maximum number of messages in transit on a link and ii) the counter size is larger than the maximum number of messages that can be sent (by the underlying protocol) during one execution of the deadlock detection protocol. The second assumption is one that theoreticians are often reluctant to make because it involves giving up a purely asynchronous model. But in practice, even in the largest and fastest of networks, a 64 bit counter should meet both conditions. The upshot is that we believe that deadlock algorithms that rely on counters can be made stabilizing, at least for all practical purposes.

6 Counter Flushing on Trees: Propagation of Information with Feedback

In Propagation of Information with Feedback (PIF), let us assume a single leader wishes to broadcast some information value to all nodes in the network and wishes to know when the information has reached all the nodes. In the stabilizing setting, we assume that the leader has a stream of values it wishes to broadcast to all neighbors; only after the i-th value is broadcasted to all nodes is the $i+1$-st value broadcasted. We model this by having the leader 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 appplication. However, we prefer not to model this and concentrate on the application of counter flushing to PIF.

We will assume as usual that we have a leader node (say r) and a spanning tree rooted at node r, 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 message 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 simply 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 (i.e., the leader) receives an ack from all children, the root starts a new cycle by choosing a new value.

To make the protocol stabilizing, we will tag each message sent and each value stored with a counter. When sending a new value, the root chooses a new counter value. Nodes accept a new value only when it is tagged with a different counter value from the counter stored at the node. Nodes accept acks only when the counter in the ack matches their current ack value. The correctness follows from the usual counter flushing argument. The code for the protocol is given in Figure 6.

Another fairly general method for constructing stabilizing protocols is the method of local checking as described in [APV91] and [Var92]. In fact in [Var92] there is a theorem that states that any locally checkable protocol on a tree can be stabilized using local checking. Thus it is natural to ask whether we can solve the stabilizing PIF problem with local checking instead of counter flushing. However it is easy to show that the PIF protocol of Figure 6 is not locally checkable. A protocol is locally checkable only if whenever every pair of neighbors is in a good state, then the system is in a good state. Suppose we find a bad global state of a protocol such that every pair of neighbors is in a state that appears in some other good global state. Then every pair of neighbors appears to be in a good state locally but the system is in a bad state, and hence the protocol is not locally checkable.

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 6 is not locally checkable.

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], distributed infimum[13] 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 6 appears to be important because it appears to offer a stabilizing solution to many problems that require total solutions.[14]

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

[14]Tel also distinguishes between centralized solutions with a single initiator and decentralized solutions with multiple initiators. However, in a stabilizing setting with multiple phases of the total algorithm, decentralized

We assume all counters are integers in the range $0..Max$ and all values are drawn from some domain V.
A token message 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 r.

```
Finished(i) (* boolean function used by action routines below *)
  (*set to true when not expecting tokens from any children *)
  Return true if for all children k of i: token_expected_i[k] = false

ROOT_START_r (* Leader starts a new cycle of broadcasting values *)
  Preconditions:
    Finished(r)
  Effects:
      v_r = f(v_r) (* compute new value to be broadcast*)
      count_r = CHOOSE(Max, c) (*choose new counter value*)
      For all children k of r
        token_expected_r[k] = true (* set to true when expecting a token*)

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 ≠ parent(i) or (j = parent(i) and Finished(i))

RECEIVE_{j,i}(Token, c, v) (* Node i receives token from Node j *)
  Effects:
    If j = parent_i and c ≠ count_i then (* new counter from parent *)
      v_i = v (* set stored value equal to value in token messge*)
      count_i = c (*set local counter to counter in token message*)
      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

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

Figure 6: Code for Stabilizing Propagation of Information with Feedback

The reader may feel that the 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 counter example 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.

We note that another interesting application of the stabilizing PIF protocol is for topology update. For example in the Autonet [MAM+90], topology distribution is done over a tree.

7 Counter Flushing in General Graphs: A Simple Reset Protocol

We now broaden the scope of counter flushing to consider general graphs. We continue to assume that we have a leader r that is the root of a BFS spanning tree. However, 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 very simple. As before a nodes i only accepts a new counter value c from its parents 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 message on any cross links it is part of, and waits to get a token (numbered with c) before it sends an token to its parent.

The only tricky part of the protocol is to decide how to reply to token messages received on cross links. Before we see what the problems are, let us introduce an application for this general counter flushing paradigm. Suppose we have an underlying protocol P and suppose that the leader may perioidically 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 messages received by node j in j's reset interval must be a proper prefix of the sequence of messages 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 and has sent a token message containing 5 to j. Node j has not received information on the

solutions appear to be infeasible without some sort of coordination among the initiators between phases.

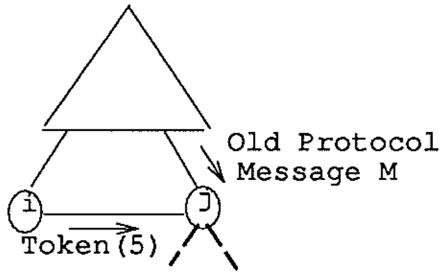


Figure 7: Reason for delaying responses to token messages received on cross links.

current reset and has an "old" protocol message in transit from its parent. Suppose node i's token message 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 message and send another "old" protocol message to node i. Thus we could have a message 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 acccept the new counter value) when it receives the token message 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 [KP90]. Katz and Perry resolve this livelock problem by having each token message carry a counter *and* a list of visited nodes; a token message is dropped when it visits a node that is in its list of visited nodes. While this solution works, it greatly increases both the message and time complexities of the solution.

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 messages with the counter at the sending node; we discard a protocol P message with counter that does not match the receivers counter. While this solution eliminates the problem in Figure 7 because the "old" message will have a different counter value from that of node i, it introduces another problem. Suppose node i sends a protocol P message to j after node i resets, but the message is received before j resets. Then if we simply check the message tag, the message will be dropped at j. One might consider buffering the message at j if the counter tag in the message 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 message received . Thus in Figure 7 when j receives the token message 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 message numbered 5 from its parent. We also do not allow protocol P to send messages at node i if node i is waiting for a token messages on one of its links. This implies that (in good executions) any message sent by i after i has locally reset is sent after j is at the same counter value as i; thus this message will be accepted by j.

7.1 Proof of Reset Protocol

Intuitively, a good "home state" for the reset protocol of Figure 8 is one in which the reset has terminated and all nodes have the same value of the counter (say c), no node is expecting a token, and all token messages in transit have value c.

Thus define a *home state with value c* of the reset protocol to be a global state in which i) For all nodes i, $Finished(i) = true$ and $count_i = c$ ii) If there is a $(Token, c')$ message stored in any link, then $c' = c$.

Define the round trip delay R through the tree to be equal to $4(h+1)$ time units, where h is the height of the tree. For a BFS tree, the height is $O(D)$ where D is the network diameter. (Roughly, R is the maximum time to send a message from the leader to a leaf node and have it acknowledged at the leader. Note that it takes 2 units of time to send a message on a link to a receiver: one unit of time for the SEND event to occur at the node, and 1 unit of time for the link to deliver the stored message.)

The following lemmas and theorems are of the form: if property X remains true for t time, then Property Y will hold after t' time. In a purely asynchronous environment where locally controlled actions are only assumed to eventually occur, the results easily translate to the form: if property X holds true indefinitely, then property Y will eventually occur. We prefer the timed version of our results since they typically provide more information and because we believe that stabilization time is a crucial measure.

Our first lemma, states that if the leader's counter remains fixed for R time units, then the protocol enters a home state.

Lemma 7.1 *If starting from any state, $count_r$ remains at c for R time units, then at the end of this interval the reset protocol will enter a home state with value c.*

A token message 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 that $parent(i)$ points to i's parent in the tree and the root is node r

$Finished(i)$ (* 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 $token_expected_i[k] = false$

REQUEST_RESET$_r$ (* root receives request to reset Protocol P *)
Effects:
if $Finished(r)$ then (* ignore request if finishing current reset *)
$count_i$ = CHOOSE(Max, c)
LOCAL_RESET(r) (* locally reset Protocol P *)
For all neighbors k, $token_expected_r[k] = true$

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 $Finished(i)$)

RECEIVE$_{j,i}(Token, c)$ (* Node i receives token from Node j *)
Effects:
If $j = parent_i$ and $c \neq count_i$ then (* counter matches node counter *)
$count_i = c$ (* set value to counter in token message*)
LOCAL_RESET(i) (* locally reset Protocol P *)
For all neighbors $k \neq j$ of i
$token_expected_i[k] = true$ (* set to true when expecting a token message *)
Else if $count_i = c$ then
$token_expected_i[j] = false$

RESET_FINISHED$_r$ (* root reports finishing of reset *)
Preconditions:
$Finished(r)$

Protocol P messages are only sent at node i when $Finished(i)$ is $true$ and are tagged with $count_i$.
A Protocol P message 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

Proof: (Sketch) It is easy to show by induction that within time $2h_i$ each node with height h_i will set $count_i = c$ and the value of $count_i$ will remain unchanged till the end of the interval. (Intuitively this is because each node accepts any value sent to it by its parent in the tree and each node will retransmit 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. In time $2h + 2$, each node will receive a token message 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 messages will have counter value c, and $token_expected_i[j] = false$ for all "cross" links (i, j). It is easy to see that within $2h$ time units after such a state tokens flow up the tree and result in a home state. □

We define a *valid request* to be a REQUEST_RESET action that occurs in a state in which $Finished_r = true$. (Intuitively, this is a reset request that is accepted as a new request; invalid requests are ignored.)

Lemma 7.2 *Within R time of starting from any state, either a valid request event occurs or the reset protocol enters a home state.*

Proof: We know from the code that the only event that can change $count_r$ is a *valid request* event. If a valid request event occurs in R time, we are done; but if not, $count_r$ remains fixed for R time and so we are done by Lemma 7.1. □

We assume that we use the Random-Increment choice function, where the random component of the counter has size $> 8(L_{max} + 1)m$, where m is the number of links and L_{max} is the maximum number of values that can be stored on a link; as usual the determinsitic component is assumed to have size $> (L_{max} + 1)m$. It is easy to see that within a constant expected number of iterations of the CHOICE function, a fresh value is chosen. It is quite easy to believe that the choice of a fresh value will result in the protocol entering a home state (see Theorem 7.5 ahead which is the basic stabilization theorem). However, to prove this simple fact we need a number of simple definitions and lemmas.

7.1.1 Fresh Counters and Fresh Counter Intervals

We say that a global state s is *fresh* with value c if:

- $count_r = c$
- $count_i \neq c$ for all $i \neq r$

- for any $(Token, c')$ message on any link, $c \neq c'$.

Define a *fresh counter interval* to be an execution fragment[15] E such that:

- The first action in E is a valid reset request.
- The second state (i.e., the state following the reset event) is fresh.
- The last state in E is the first state in E (other than than the first state) in which $Finished(r) = true$.

The value of interval E is the value of 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 $count_j = c$. (intuitively $I(j)$ is the index of the first state in which node j is initiated into the current reset computation.)
- Let $F(j, k)$ be the index of the first state such that $count_j = c$ and $token_expected_j[k] = false$ (intuitively $F(j, k)$ is the index of the first state in which node j has been initiated into the current reset computation and also knows that node k has been initiated into the current reset computation.)
- Let $F(j)$ be the index of the first state such that $count_j = c$ and $token_expected_j[k] = false$ for all neighbors k of j. (intuitively $F(j)$ is the index of the first state in which node j has finished the current reset computation)

We now state a cluster of simple facts relating these definitions:

Lemma 7.3 *For any fresh counter interval $[I, F]$ the following facts hold:*

- *For every pair of neighbors j and k, the states $I(j)$, $F(j)$ and $F(j, k)$ exist.*
- *If $k = parent_j$ then the interval $[I(j), F(j)]$ is contained in $[I(k), F(k)]$. (i.e., a child is initiated after its parent and finishes before its parent)*

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

- *In the interval $[I(j), F]$: $count_j = c$ (i.e., the value of a node's counter remains unchanged from the time it is initiated till the end of the interval).*
- *For every pair of neighbors j and k, $I(k) < F(j,k) \leq F(j)$ (i.e., a node cannot finish until each of its neighbors is initiated.)*
- *For any j, in the interval $[F(j), F]$: $count_j = c$ and for all neighbors k of j: $token_expected_j[k] =$ false. (i.e., node j does not change its state from the time it finishes till the end of the interval.)*
- *For every pair of neighbors j and k, in the interval $[F(j,k), F]$, any $(token, c')$ message stored in link (k, j) has $c' = c$ (i.e., after j knows that k is initiated, all token messages sent from k to j carry counter value c)*
- *$F = F(R)$ (i.e., when the root finishes the fresh counter interval finishes).*

Proof: A rough operational argument can be made using the following observation. In the initial state I, for any link (j, k) the value c is not stored in the link because c is fresh. Similarly the value c is not in any node other than the root. Thus the value c can only be stored on the link and delivered to k after j changes its value to c (i.e. in state $I(j)$). Since children only accept new counter values from their parents, a child can only be initiated after its parent is initiated. Also all further tokens received from the parent must carry the value c (by induction on height of tree). Similarly, the state $F(j, k)$ must occur after the state $I(k)$ because any token numbered c sent on this link must must have been sent after k was initiated. □

Corollary 7.4 *The last state of a fresh counter interval is a home state.*

Proof: Follows immediately from the facts in Lemma 7.3. We know that $F = F(r)$ and that $F(i) < F(r)$, for $i \neq r$. Thus all nodes i have $count_i = c$ and $token_expected_i[j] = true$ for all neighbors j (from the fifth fact). Also for all pairs of neighbors k, j, we know that $F(j,k) \leq F(j) \leq F$; hence by the sixth fact all token messages on link (j, k) have counter value equal to c. □

7.1.2 Tying Up Loose Ends: Stabilization and Correctness

In what follows, we assume that the spanning tree used is a BFS (Breadth First Search) tree and so the round trip delay $R = O(D)$, where D is the network diameter. We first show that the reset protocol stabilizes quickly.

Theorem 7.5 *Within $O(D)$ expected time of starting from an arbitrary state, the reset protocol enters a home state.*

Proof: From Lemma 7.2, if the protocol does not enter a home state every $O(R)$ units of time a valid request occurs which causes the Random-Increment function to be invoked. But in constant expected number of such invokations, the leader will choose a "fresh" counter value that is not present in the network. Once this happens, we begin a fresh counter interval. From Lemma 7.2, within $O(R)$ time of the start of the interval, we must either enter a home state or have a valid reset request. But in the latter case, we know that within $O(R)$ time of the start of the fresh counter interval there is a state in which $Finished_r = true$. But this implies that the fresh counter interval has ended in $O(R)$ time, and we know from Corollary 7.4 that the last state of a fresh counter interval is a home state. The theorem follows as $O(R) = O(D)$. □

The next theorem shows that once the reset protocol is in a home state, it behaves correctly till the next reset request.

Lemma 7.6 *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.*

Proof: Easy to see from the code and the definitions of a home state and a fresh counter interval. Notice also that when the reset protocol is in a home state, it is impossible for a node j to receive a $(Token, c)$ message with $c \neq count_j$; thus (from the code) j will never perform a local reset. □

This leads us to our last theorem: this states that any reset request that occurs after the reset protocol reaches a home state will result in successfully resetting the underlying protocol P.

Lemma 7.7 *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 $O(D)$ time after this reset request and in this home state, the underlying protocol P is in a legal state.*

Proof: We know that any reset request that begins in a home state will result in the leader picking a fresh counter value, say c, which begins a fresh counter interval. We know from Theorem 7.5 that within $O(R) = O(D)$ time, this fresh counter interval will end. Thus from Lemma 7.3 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 messages received by k from j during the interval $[I(k), F]$ is the a prefix of the sequence of messages sent by j during $[I(j), F]$. Recall that the interval $[I)j), F]$ is what we have previously called the reset interval at node j.

So consider any message m sent by j during the interval $[I(j), F]$. From the protocol code, we know that j does not send any message 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 or stored on the link. If m is delivered, m must have been delivered after $F(j)$ (since it was sent after $F(j)$) and hence by Lemma 7.3 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 the link, we know (because the link is FIFO) that all messages sent after m are not delivered. Thus, applying this argument to all messages sent by j during $[I(j), F]$, we see that: if m is received and accepted, then all messages sent before m in $[I(j), F]$ are received and accepted; but if m is not received then all messages sent after m in $[I(j), F]$ are not received.

All that remains is to show that any Protocol P message 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 messages 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 $count_j = c$.) □

In summary, Theorem 7.5 shows that the reset protocol enters a home state in $O(R)$ time. Finally, Theorem 7.6 and Theorem 7.7 show that once the reset protocol enters a home state, the reset protocol behaves correctly.

In the next section, we will consider a diffusing computation P which is initiated by the root; nodes other than the root begin to participate in the current computation only after receiving a message from some other node, and the first set of messages is sent by the root. For this special case, the reset protocol is simpler. For instance, we do not need to tag Protocol P messages with the current counter value since no such message is sent during a fresh counter

interval. Define an *initial state* of P to be a state in which each the local state of protocol P at each node (including the root) is locally reset and there is no Protocol P message on any link. Then we have the following corollary of Theorem 7.7.

Corollary 7.8 *Consider any diffusing computation P and any reset request that occurs when the reset protocol is in a home state. Then the reset protocol will enter a home state in $O(D)$ time after this reset request and in this home state, the underlying protocol P is in an initial state.*

Proof: Follows directly from Theorem 7.7, the definition of a diffusing computation, and from Lemma 7.3 which states that $F(r) >= F(i)$ for all nodes i, (i.e., the root finishes its reset interval after every other node has finished its reset interval). □

7.2 Comparison with Other Reset Protocols

To make our reset protocol a full-fledged reset protocol, we also need to augment it to allow any node to make a reset request. This is easily done by having each node having a reset request bit that is set when the node gets a reset request or when it has received a *Request* message from its children; when a node's request bit is on, it periodically sends a *Request* message to its parent. When the root gets a *Request* message, 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* messages while *Finished*(i) = *false*.

Once this is done, the resulting reset protocol looks remarkably like a message passing version of the Arora-Gouda [AG90] reset protocol. However, there are some interesting differences. First, the current versions of [AG90] are based on a shared memory model; in the high-atomicity shared memory model the size of the node counters is only required to be 2; in a shared memory model with read-write atomicity, the size of the node counters is only required to be 4. We believe that this is because the paradign underlying the stabilization of [AG90] is local checking and correction. In fact, we have shown how to derive the high-atomicity version using local correction in [AGV92], and we believe the arguments will extend to the read-write version.

In a message passing model, where each link can store L_{max} counter values, we believe that the Arora-Gouda protocol will need larger values of the counter size; however, we conjecture that using local checking the counter size can be reduced to $O(L_{max})$[16] as opposed to $O(L_{max}m)$

[16]Intuitively, this is the size required to do counter flushing on a single link, which is required for local checking to work correctly on a link

which is what is required by counter flushing. However, we believe that implementations using counter flushing are simpler to implement and understand than those using local checking; the cost of this simplicity is very slight: an increase in the counter size, which in practice is hardly an issue.

Another subtle 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 [AG90], 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 because the memory model is intuitively a "pull" model (nodes read the states of their neighbors) as opposed to a message passing model which is a "push" model (nodes send their states to their neighbors). Thus we conjecture that some modification like ours is required to deal with this problem.

Finally, there is the stabilizing reset protocol of [APV91] which is in turn based on the non-stabilizing (and classic) AAG 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 this protocol faster by running it over a spanning tree but in that case much of the complexity of that protocol is not needed[17] and there is no advantage to using the AAG protocol over the protocol described in this paper.

8 Periodically Restarting a Terminating Diffusing Computation

Consider a computation P that is initiated by the root. We say that P is terminating if when P is started in a good state (i.e. all nodes are initialized and the links are free of protocol P messages) then there is some later action at the leader (say D_r) which signifies that the computation is complete. We also require that P has what we call the *stabilizing termination* property – i.e., regardless of what state P starts in, in a bounded amount of time, the event D_r will occur. P need not be stabilizing – the event D_r is allowed to occur even when P is in a bad state; we only require that it does occur.

Given that P has the stabilizing termination property, we use the simple reset protocol to periodically "restart" protocol P. Thus even if the initial executions of P are incorrect, after the reset protocol correctly resets protocol P, future executions of P will be correct. We demonstrate the applicability of this paradigm by applying it to the Chandy-Lamport snapshot protocol; we first augment the Chandy-Lamport protocol slightly to make it have

[17] the list of local predicates to be checked for the protocol is quite large [Var92]

the stabilizing termination property. We then apply our method to it to produce an efficient stabilizing snapshot protocol.

Compared to the stabilizing snapshot protocol of [KP90] our snapshot protocol is much faster ($O(D)$ expected time versus $O(Nr)$ time to stabilize). While other fast snapshot protocols do exist (for example [AKM$^+$93, AG90]) they are *blocking* snapshots that block the underlying application protocol. In any practical setting, blocking the application protocol for up to $O(D)$ time is infeasible; thus non-blocking snapshot protocols are important for practical applications. Note our snapshot protocol has three levels of hierarchy: at the bottom we have an application protocol A; next we have a snapshot protocol P that attempts to periodically take a snapshot of the application A; finally, on the highest level we have a simple reset protocol that periodically *resets the snapshot protocol* P. Note that the reset does not reset the application protocol A but only the snapshot protocol P.

8.1 Periodic Restart Paradigm

The basic idea of the paradigm is very simple. The leader alternates between reset phases (which resets protocol P) and phases of the protocol P. When protocol P terminates (which it is guaranteed to because of the stabilizing termination property) at the leader, the leader initiates a reset phase. When the reset protocol terminates (which is guaranteed because we use a stabilizing reset), the leader initiates a fresh version of protocol P. The alternation is accomplished by a variable *turn* at the root that can take only one of three values: *wait* (i.e., waiting for the reset to complete), *reset* (i.e., waiting for the snapshot protocol to complete to initate the next reset) and *not_reset* (i.e., waiting to initiate the next instance of protocol P). When the turn variable is *reset* and Protocol P appears to have terminated, the root node makes a reset request and changes *turn* to *wait*. When the reset protocol terminates (as signalled by a RESET_FINISHED action), *turn* is changed to *not_reset*. When the turn variable is *not_reset* and Protocol P appears to have terminated, the root node makes initiates a fresh version of Protocol P.

After a successful reset phase, it is easy to see that all nodes are initialized correctly for protocol P and that there are no protocol P messages outsanding. Thus the next protocol P phase will work correctly and provide a correct answer. Suppose that protocol P is a deterministic protocol that always computes the same result (e.g., a unique minimum spanning tree). Then we can keep two sets of output variables for protocol P; one set of variables is used as a scratchpad and is updated by the current computation of protocol P; the other set of variables is used as the actual output variable. At the start of the reset phase, the protocol can copy the results in the scratchpad variables to the output variables. It is easy to see that

if P is a deterministic, terminating protocol then within bounded time, the output variables of protocol P will converge to correct values.

This method is also applicable if each computation P is potentially different from other computations of P, and the outputs on each computation are completely different. A simple example of this is the Chandy-Lamport snapshot protocol ([CL85]) which we discuss below.

8.2 Example of Periodic Restart: Fast Stabilizing Snapshot

We consider the Chandy-Lamport snapshot protocol ([CL85]) and especially its application by Katz and Perry ([KP90]) to compile arbitrary distributed programs into stabilizing equivalents. In the method of Katz and Perry, the leader repeatedly invokes the snapshot procedure to check whether the application protocol A is in a good state. Katz and Perry show how to make the Chandy-Lamport (CL) protocol stabilizing such that within bounded time, correct snapshots are produced. As we said before, the stabilization time of their stabilizing snapshot is quite large ($O(n^2)$, where n is the number of nodes). So our stabilizing snapshot provides essentially a replacement snapshot protocol that is also non-blocking but is faster ($(O(D)$ expected time). This improves the efficiency of the compiler in [KP90].

In the normal CL protocol, the leader sends a special marker message to all neighbors and sets a flag called *mark_expected* for each link, to indicate that it expects a marker back on this link. Nodes other than the leader that receive a marker for the first time do the identical procedure except that they set the *mark_expected* flag to false for the link they received the marker flag from. A node receiving a marker on a link for which *mark_expected* is *true* sets *mark_expected* to *false*. The actual snapshot consists of i) for each node i, the state of node i at the instant node i receives the marker for the first time and ii) for each link (i,j), the sequence of messages received by node j while node j had the *mark_expected* flag set for link (i,j).

To make the Chandy-Lamport protocol have the stabilizing termination property we do the following. First, we use the tree to report the results of the snapshot up the tree. When a leaf has collected its part of the snapshot, the leaf sends this information up to its parent. When interior nodes have reeived snapshot information from their children, they merge this information with their own snapshot information and send it up to their parents. Eventually the root obtains the resulting snapshot and the process terminates.

There are several potential sources of deadlock that must be corrected for. First, in order to ensure that the *mark_expected* flags at a node are eventually cleared, each node i periodically retransmits Marker messages on link (i,j) while the snapshot procedure is executing. Second,

each non-leaf node keeps a flag *snap_expected* for all its children in the tree which is set to true at the instant a node receives a Marker message for the first time and is set to false when the node receives snapshot information from its child. To prevent deadlock, a node that has acquired snapshot information (from all its children and itself) periodically sends this snapshot information to its parents.

Each node i has a variable s_i that holds its current snapshot information and d_i which holds cumulative snapshot info received from children. Each node snapshot record s_i is is a 3-tuple containing a node ID, a state for the node itself and a set of link records for each of its links. Each link record is itself a tuple containing an identifier of the link and a sequence of messages. The cumulative snapshot information is a set of node snapshot records. We assume these variables are large enough to hold the largest possible snapshots for each node.[18]

Finally, besides the normal protocol variables, each node i keeps a flag $busy_i$. When this variable is cleared, node i is not executing the snapshot protocol. In particular, node i refrains from sending snapshot messages to its parent when $busy_i = false$. Since $busy_i$ is cleared by the reset procedure, this prevents node i's parent from receiving results of the previous snapshot while the current snapshot is in progress. Similarly when $busy_i = false$, node i will not transmit marker messages. The actual CL snapshot protocol begins executing at node i when i receives a marker while $busy_i = false$. Thus in a correctly working snapshot phase, the first marker sent on each link corresponds to the markers sent by the original CL protocol.

To control invocations of the reset protocol, the root r also keeps a *turn* variable. When the local variables at the root indicate that the snapshot has completed and the *turn* variable is equal to *reset*, the root requests a reset and sets its state to *wait* indicating that it is waiting for the reset to complete. When the reset procedure indicates that the reset has terminated, the root sets the *turn* variable to *not_reset*. Finally when the local variables at the root indicate that the snapshot has completed and the *turn* variable is equal to *not_reset*, the root initiates a new snapshot.

The code for the snapshot protocol is described in Figure 9 and Figure 10.

8.3 Proof of Snapshot Protocol

We only quickly sketch the major lemmas and theorems. By $O(D)$ we mean that there is some natural constant k such that the worst-case time is $\leq kD$. where D is the network diameter.

[18] In practice, it may be difficult to bound the size of the link records without making some assumptions about time or about the particular application protocol.

A marker message is encoded as *Marker* and a snapshot message containing snapshot info as $(Snapshot, d)$
The state of each node i consists of:
a boolean variable $busy_i$, a variable $turn_i \in \{reset, wait, not_reset\}$,
boolean flags $mark_expected_i[j]$ for every neighbor j and $snap_expected_i[k]$ for every child k.
Variables s_i (current snapshot info) and d_i (cumulative snapshot info from children).
We assume that $parent(i)$ points to i's parent in the tree and the root is node r

SNAP_FINISHED(i) (* macro used
Return true if both the following conditions are true
For all children l: $snap_expected_i[l] = false$
For all neighbors k: $mark_expected_i[k] = false$

REQUEST_RESET$_r$ (* output action, requests a reset of snapshot protocol state *)
Preconditions:
$turn_r = reset$ (* reset protocol's turn to run *)
SNAP_FINISHED(r)
Effects:
$turn_r = wait$ (* indicates a waiting for snapshot to complete *)

RESET_FINISHED$_r$ (* input action reports that current reset is over *)
Effects:
$turn_r = not_reset$ (* snapshot protocol's turn to run *)

START_SNAPSHOT$_r$ (* output action, Root initiates snapshot *)
Preconditions:
$turn_r = not_reset$ (*snapshot protocol's turn to run *)
SNAP_FINISHED(r)
Effects:
$turn_r = reset$
Initialize d_r, s_r to be empty. Record current state in s_r
$busy_r = true$
For all children l
$snap_expected_r[l] = true$
For all neighbors k
SEND$_{r,k}$(*Marker*)
$mark_expected_r[j] = true$

Figure 9: Variables and Code Used to Initiate Snapshots and Resets

SEND$_{i,j}$(*Marker*) (* Retransmit marker*)
Preconditions:
$busy_i = true$ or $((i = r)$ and (SNAP_FINISHED$(r)))$

RECEIVE$_{j,i}$(*Marker*) (* Node i receives marker from j*)
Effects:
if $busy_i = false$ and $i \neq r$ then
Initialize d_i, s_i to be empty. Record current state in s_i
$busy_i = true$
For all children l
$snap_expected_r[l] = true$
For all neighbors $k \neq j$
$mark_expected_i[j] = true$
SEND$_{i,k}$(*Marker*)
if $mark_expected_i[j] = true$ then
$mark_expected_i[j] = false$ (* stop expecting marker from j*)
if for all neighbors k: $mark_expected_i[k] = false$ then
$d_i = d_i \cup \{(s_i, i)\}$ (* record your own state *)

SEND$_{i,j}$(*Snapshot*, s) (* Node i sends snapshot information to j*)
Preconditions:
$busy_i = true$
$parent_i = j$
$d = d_i$ (*snapped state from node i and all other nodes *)
SNAP_FINISHED(i)

RECEIVE$_{j,i}$(*Snapshot*, s) (* Node i receives snapshot from j*)
Effects:
if j is a child of i and $snap_expected_i[j] = true$ then
$d_i = d_i \cup s$
$snap_expected_i[j] = false$

LOCAL_RESET(i) (* procedure called to locally reset snapshot *)
For all children l: $snap_expected_i[l] = false$
For all neighbors k: $mark_expected_i[k] = false$
$busy_i = false$

While $mark_expected_i[j] = true$, node i records the sequence of application messages arriving from node j in the link record for neighbor j contained in s_i
All SEND actions are in a separate class that occur in 1 unit of time.

Figure 10: Code for Chandy-Lamport Snapshot Detection

Our major tool is Theorem 7.5 which states that the reset protocol stabilizes in $O(D)$ expected time *regardless* of the behavior of the client (i.e., snapshot) protocol P. This avoids circularities because the behavior of P clearly depends on the reset protocol. Note that the snapshot protocol is the combination of the snapshot and reset protocols.

The major termination theorem about the snapshot protocol is that the root will assume that the snapshot is complete (i.e., *snap_expected*(r) becomes true) in $O(D)$ expected time starting from any state.

Theorem 8.1 *Within $O(D)$ expected time of any state, there is a state in which* SNAP_FINISHED = *true.*

Proof: We only sketch the main idea. We know from Theorem 7.5 that within $O(D)$ expected time, the reset protocol reaches a home state s. If s.SNAP_FINISHED(r) = *true* we are done, so assume that s.SNAP_FINISHED(r) = *false*. We claim that in kD time starting from s (where k is a sufficiently large constant), we reach a state in which SNAP_FINISHED(r) = *true*.

Suppose the last statement is false. Then for kD time after s, there can be no REQUEST_RESET actions as these are only enabled if SNAP_FINISHED(r) = *true*. But from Theorem 7.6 this means that the reset protocol remains in a home state for kD time. Thus the snapshot protocol at node i cannot be locally reset in this interval since such local resets can never occur in a home state.

But this implies that for some sufficiently large constant $k_1, k_1 < k$, within $k_1 D$ time after s, we reach a state s' in which all nodes $i \neq r$ have $busy_i$ = *true*. This follows by induction on the height of the tree and three facts: the root keeps sending markers down the tree when SNAP_FINISHED(r) = *false*; nodes that are not busy become busy after receiving a marker and keep transmitting markers down the tree; nodes clear their busy flag only after doing local resets, which cannot occur.

But this implies that for some sufficiently large constant $k_2, k_1 + k_2 < k$, within $k_1 D$ time after s', we reach a state s'' in which all nodes i have SNAP_FINISHED(i) = *true*. This follows by reverse induction on the height of the tree (i.e., starting from the leaves and moving upwards) and the following facts: in constant time after s', every node j sends a marker to every neighbor k causing $mark_expected_k[j]$ to become false; the root cannot set its *mark_expected* or *snap_expected* flags to true during this interval because this is only done in a START_SNAPSHOT action that cannot occur when SNAP_FINISHED(r) = *false*; nodes other than the root cannot set their *mark_expected* or *snap_expected* flags to true during this interval because they cannot do this until they clear their busy flag which cannot happen; each node (starting from the

leaves) sends a *Snapshot* message up to its parent in constant time after its *snap_expected* flags for all children and its *mark_expected* flags for all neighors are clear; finally a node clears $snap_expected_i$ when it has received *Snapshot* messages from its children.

In summary, in $O(D)$ expected time after s we reach a state in which SNAP_FINISHED(r) = *true*; hence in $O(D)$ time starting from any state we reach a state in which SNAP_FINISHED(r) = *true*. □

The next lemma states that the turn variable is guaranted to change its value in $O(D)$ expected time.

Lemma 8.2 *Within $O(D)$ expected time of any state, there is a state in which $turn_r$ changes value.*

Proof: We only sketch the main idea. We know from Theorem 8.1 that in $O(D)$ expected time we reach a state s in which SNAP_FINISHED(r) = *true*. So we consider cases:

- $s.turn_r = wait$. In this case, we know from Theorem 7.5 that the reset protocol will enter a home state s' in $O(D)$ expected time. If the value of $s'.turn_r \neq wait$ we are done, so assume that $s'.turn_r = wait$. Now the REQUEST_RESET action is not enabled while $s.turn_r = wait$. Thus the reset protocol will remain in a home state while $turn_r = wait$ and so in $O(1)$ time after s', a RESET_FINISHED event will occur (from the timing conditions for this event which is enabled in a home state). This will cause $turn_r = not_reset$.

- $s.turn_r = not_reset$. In this case, we know that SNAP_FINISHED(r) will remain *true* until the value of $turn_r$ changes (as SNAP_FINISHED(r) can only be set to *false* by a START_SNAPSHOT action). But the START_SNAPSHOT action is enabled whenever $turn_r = not_reset$ and SNAP_FINISHED(r) = *true* and so this action must occur in $O(1)$ time after s, resulting in a state in which $turn_r = reset$.

- $s.turn_r = reset$. In this case, we know that SNAP_FINISHED(r) will remain *true* till $turn_r$ changes (as SNAP_FINISHED(r) can only be set to *false* by a START_SNAPSHOT action and the START_SNAPSHOT action is not enabled when $turn_r = not_reset$). But the REQUEST_RESET action is enabled whenever $turn_r = not_reset$ and SNAP_FINISHED(r) = *true* and so this action must occur in $O(1)$ time after s, resulting in a state in which $turn_r = wait$.

□

Define a *good reset request* to be a $(s, \text{REQUEST_RESET}, s')$ transition such that s is a home state.

We claim that:

Lemma 8.3 *A good reset request occurs in $O(D)$ expected time starting from any state; all subsequent reset requests are also good reset requests.*

Proof: We know from Theorem 7.5 that in $O(D)$ expected time we reach a home state s of the reset protocol. We know that the reset protocol remains in a home state till a reset request occurs; thus any reset request after s ia a good reset request. We know from Lemma 8.2 that the value of the $turn_r$ variable changes twice in $O(D)$ expected time after s. But we know from the code that the $turn_r$ variable only changes from *wait* to *not_reset*, from *not_reset* to *reset*, and from *reset* to *wait*. Thus in $O(D)$ expected time after s, we must have a transition (s', π, s'') such that $s'.turn_r \neq wait$ and $s''.turn_r = wait$. It is easy to see from the code that $\pi = \text{REQUEST_RESET}$.

It is also easy to see from Theorem 7.6 and Theorem 7.7 that all future reset requests occur in a home state and hence are good reset requests. □

Define a *initial* state of the snapshot protocol to be a state in which i) for all nodes i, $\text{SNAP_FINISHED}(i) = true$ and $busy_i = false$ and ii) there is no (*Marker*) message in transit on the links. Notice that this corresponds to the definition of an *initial state* for a diffusing computation that we gave in the last section.

The next lemma follows directly from Corollary 7.8.

Lemma 8.4 *In $O(D)$ time after a good reset request, the reset protocol will enter a home state and in this home state, the snapshot protocol P is in an initial state and $turn_r = not_reset$. These three conditions will remain true until a* START_SNAPSHOT *event.*

Theorem 8.5 *Consider any state in which the reset protocol is in a home state, the snapshot protocol is in an initial state, and $turn_r = not_reset$. Then a good reset request will occur in $O(D)$ time after this state and in the state before this reset request, the variable d_r will hold a valid snapshot of the application protocol.*

Proof: It is easy to see that within $O(1)$ time, a START_SNAPSHOT action occurs, which sets SNAP_FINISHED(r) = *false*. This will begin a phase of the snapshot algorithm starting with a state in which there are no markers on links and all nodes i have SNAP_FINISHED(i) = *true*. We can map the first *Marker* messages and *Snapshot* messages sent on every link during this phase to the *Marker* and *Snapshot* messages sent by a augmented version of the Chandy-Lamport snapshot (in which snapshot information is reported up the tree). Subsequent copies of *Marker* and *Snapshot* messages received on a link do not change the state of the receiving node and can be ignored. Since the Chandy-Lamport protocol produces an accurate snapshot in d_r by the time SNAP_FINISHED(r) = *true*, so does this phase of our snapshot. Also the value of d_r cannot change until $busy_r$ becomes false; but this cannot happen till after the next good reset request. And we know from Lemma 8.3 that a good reset request occurs in $O(D)$ expected time. □

9 Virtual Circuit Protocols

We conjecture that another interesting application of counter flushing is to design stabilizing virtual circuit protocols. In a virtual circuit protocol we assume that there is a sequence of nodes Node 0 to Node m that are connected and that we wish to send data "reliably" from Node 0 to Node m. We assume that the links are reliable and FIFO links and thus the only real difficulty arises if a virtual circuit is cancelled and another virtual circuit is set up immediately afterwards; we wish to make sure that data from the first virtual circuit is not delivered as part of the second virtual circuit.

We assume that the sequence of nodes is fixed and there is at most one virtual circuit for each such fixed path of nodes. In practice, it is easy to extend this to have multiple VC's per path. In practice, too, a user at Node 0 will attempt to set up a circuit to a user at Node m. This is done by first consulting a routing protocol to obtain a unique path from Node 0 to Node m; the complete (acyclic) path could be carried in every virtual circuit control message. Thus once the routing protocol has stabilized, a virtual circuit from Node 0 to Node m will pass through a fixed sequence of nodes. However, we ignore this detail and simply assume that the sequence of nodes comprising the virtual circuit is fixed.

Since we wish to "flush" out data from any previously set up virtual circuits, it is natural for the set up of a virtual circuit to follow the sequence shown in Figure 11. When setting up a virtual circuit, the source sends a set message S down the path. In a correctly set up virtual circuit, the set message propagates to the destination node m after which it is reflected (Figure 11) to the source; only then can data be sent and received correctly because the set

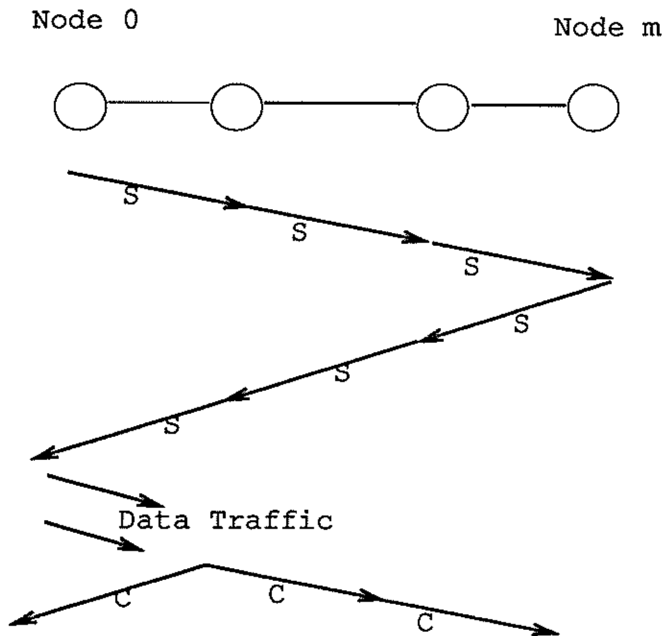


Figure 11: Example of virtual circuit setup. The S messages are set up messages and the C messages are cancel messages.

messages have "flushed" the path of old data messages. Nodes along the path can, however, cancel the circuit at any time by sending a cancel message backwards and forwards alongthe path (Figure 11).

In Spinelli's thesis ([Spi88]), Spinelli shows that if a node simply sends a cancel message both backwards and forwards, then it is possible to incorrectly set up a virtual circuit. Spinelli modifies the simple-minded cancellation shown in Figure 11 as follows. A node sending a cancel message toward the destination does not go into a disconnected state immediately but goes into a "conditional disconnect" state where it waits to get an ack back from the next node in the sequence which indicates that the next node has received the cancel. Nodes receiving a cancel message from a forward neighbor (i.e., one that is closer to the source) also send a cancel ack back to the forward neighbor.

The main reason for node i to send a cancel message to node $i+1$ is to flush out spurious set up messages that may be lurking in the channel from $i+1$ to i. If these are not flushed out, and if i quickly receives a set up message from node $i-1$ (i.e., immediately after the previous cancellation an attempt is made to restablish the circuit) then i may wrongly set up the second circuit.

However, suppose we use counter flushing. Thus all set up requests and cancel messages

carry a counter value and each node includes a counter value as part of its state. Node i accepts new set up requests from node $i-1$ only if the counter value stored at node i is different from the counter value in the message; node i accepts a set up message from node $i+1$ only if the the counter value stored at node i is the same as the counter value in the message. Thus it appears that if node 0 uses a "fresh counter" value to set up a new virtual circuit, then the new virtual circuit will be set up correctly according to the structure shown in Figure 11[19]. Thus it appears that cancellation can follow the simple structure shown in Figure 11 without the need for a Cancel Ack.

We have constructed a preliminary version of a stabilizing virtual circuit protocol by applying counter flushing to a somewhat simpler version of [Spi88]. We leave the details and proof of this protocol for future work.

10 Conclusions

We make the following remarks about the counter flushing paradigm:

- **Wide Range of Applications and Topologies:** We have shown that counter flushing is a powerful paradigm for designing stabilizing protocols by exhibiting stabilizing protocols for token passing on a ring and in general networks, broadcast with feedback on trees, deadlock detection and reset in arbitrary networks. For counter flushing to be applicable we do need to elect a leader and (in general networks) compute a BFS tree rooted at the leader; however, this can be done efficiently using the protocol of [AKM+93]

- **Underlying theme to applications:** Besides being a paradigm that helps us design new stabilizing protocols, counter flushing is also a *unifying* paradigm. Part of the unexpected pleasure of writing this paper was realizing the connection between seemingly disparate protocols such as Dijkstra's token ring protocol, Afek and Brown's Data Link protocol, Misra's deadlock detection protocol, Arora and Gouda's reset protocol, and Spinelli's virtual circuit protocol. 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, Virtual Circuit, and Reset problems can be regarded as *synchronization* problems whose correctness can be formalized in terms of a mating relation [AE83, Spi88, Var92]

[19] assuming that no nodes cancel while this circuit is being set up; this can be enforced by ensuring that cancellation is never done during circuit set up

- **Easy to Design and Prove:** To add counter flushing to an appropriate application, we: i) first add counters to all nodes and messages ii) add actions to ensure that the protocol does not deadlock and the leader will change its counter in $O(D)$ time. The use of the appropriate CHOICE function then ensures that the protocol will pick a fresh counter value in $O(D)$ expected time iii) By carefully restricting the way new counters are accepted (e.g., from parents only) we ensure that a fresh counter value succeeds in "flushing" the network of old counter values and that at the end of such a fresh counter interval the protocol is in a good state.

- **Competitive with Local Checking:** Local Checking and Correction is another general paradigm that we have used before ([APV91, Var92, AGV92]) 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., leader election). 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. Our experience in [CSV89] indicates that it takes some care to implement local checking; when there is a large amount of local state, one has to either slow down the normal protocol during local checking or have support to do a fast local snapshot. Local checking also requires a careful enumeration of the protocol's local predicates, which can be quite large even for a moderately complex protocol like a reset. [APV91, Var92].

Finally, regardless of counter flushing, we believe that the paradigm of periodically restarting a diffusing computation P with the *stabilizing termination* property is of independent interest. As in the counter flushing paradigm, it is possible to add extra actions to a protocol P to make it have the stabilizing termination property; this in turn can be used to provide a version of P that stabilizes in $O(D + T_P)$ expected time where T_P is the time complexity of protocol P.

We have used this to produce the first stabilizing non-blocking snapshot protocol that has $O(D)$ expected stabilization time. This in turn considerably improves the efficiency of the general compiler of [KP90] so that any arbitrary protocol can now be stabilized in $O(D)$ expected time. Note that while this seems to subsume most previous results, there is another metric which is roughly the message complexity ([Var92]) of checking. While the improved Katz and Perry compiler is fast, it still has a rather large message complexity. Thus the search continues for less general but more efficient stabilization techniques: we believe the counter flushing paradigm of this paper is one such technique.

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.

[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.

[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.

[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.

[AGV92] Anish Arora, Mohamed G. Gouda, and George Varghese. Distributed constraint satisfaction. Unpublished manuscript, February 1992.

[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.

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

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

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

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

[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.

[CM81] K. Mani Chandy and J. Misra. Asynchronous diistributed simulation via a sequence of parallel computations. *Comm. of the ACM*, 24(4):198–205, April 1981.

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

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

[DIM89] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, 1989. Also, available as MCC Technical Report No. STP-379-89.

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

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

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

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

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

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

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

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

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

[Mis83] Jayadev Misra. Detecting termination of distributed computations using markers. In *podc2*, Montreal, Ontario, August 1983.

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

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

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Trans. on Info. Theory*, IT-29(1):23–35, January 1983. Some details in technical report of same name, MIT Lab. for Info. and Decision Syst., LIDS-P-1015; Technion Dept. EE, Publ. 414, July 1981.

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

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

[Var92] George Varghese. Self-stabilization by local checking and correction. Ph.D. thesis, to appear as a LCS Technical Report, 1992.