Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols

EXTENDED ABSTRACT

Baruch Awerbuch* George Varghese†

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We explore the notion of distributed program checking as a means of making a distributed algorithm self-stabilizing. We describe a compiler that converts a deterministic synchronous protocol π for static networks into a self-stabilizing version of π for dynamic networks. Let T_π be the time complexity of π and let D be a bound on the diameter of the final network. The compiled version of π stabilizes in time $O(D+T_\pi)$ and has the same space complexity as π. Our general method achieves efficient results for many specific *non-interactive* tasks. For instance, our solutions for the shortest paths and spanning tree problems take $O(D)$ to stabilize, an improvement over the previous best time of $O(D^2)$. We provide improved solutions for many other problems including topology update, minimum spanning trees, colorings, maximum flows, and maximal independent sets.

Keywords: Fault-tolerance, Synchronous Compilers, Self-stabilization, Distributed Checking

1 Introduction

Consider the following door closing protocol. A person going through the door shuts the door when she leaves. Correct behavior – especially important on a cold day – requires that the door be shut when not in use. If the door is initially closed, then the door closing protocol maintains correct behavior. If however, the door is initially open, or some user has made a mistake, then the door can remain open forever. Thus the door closing protocol is not self-stabilizing - i.e., it does not work correctly when its initial state is incorrect. It is often a good idea to make the door closing protocol self-stabilizing by adding a spring that constantly restores the door to the closed position.

A distributed network algorithm consists of a program for each network node. Each program consists of code as well as local state. The global state of the algorithm consists of the local state of each node as well as the messages on network links. We define a catastrophic fault as a fault that arbitrarily corrupts the global network state, but not the program code.

Self-stabilization formalizes the following intuitive goal: *despite a history of catastrophic failures, once catastrophic failures stop, the system should stabilize to correct behavior without manual intervention.* Self-stabilization is a reasonable and interesting model because:

Catastrophic faults occur: Code[1] can be protected by redundancy since it is never modified. However, state is constantly being updated. Thus it is hard to infallibly protect state from memory corruption. It is also hard to prevent a malfunctioning device from sending out an incorrect message. A self-stabilizing model subsumes most previous models of network faults. For instance, catastrophic faults clearly go beyond well-studied fault models such as node and link failures.

Manual intervention has a high cost: In a large decentralized network, restoring the network manually after a failure requires considerable coordination. As in

*Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

†Supported by the DEC Graduate Education Program.

[1] While input is often static (as in the case of node IDs), we deal with changing input by requiring that such input be the output of another self-stabilizing protocol. For example, the list of neighboring nodes in a dynamic network can be supplied by a self-stabilizing Data Link protocol.

the case of the AT&T network, the consequent network shutdown has a large dollar cost.

These issues are illustrated by the crash of the original ARPANET protocol ([Ros81] [Per83]). The designers used a sequence number to distinguish newer topology updates from older ones. Because the sequence number was finite, they used a circular number space. Hence, it is possible to have three sequence numbers a, b, c such that $a > b > c > a$. The protocol was carefully designed to never enter a state that contained three such updates. Unfortunately, a malfunctioning node injected three such updates into the network and crashed. After this the network cycled continuously between the three updates. It took days of detective work [Ros81] before the problem was diagnosed. This would have been unnecessary if the protocol had been self-stabilizing.

Distributed program checking: An elegant theory of self-testing and self-correcting, pioneered by Blum [Blu], has been developed in the area of sequential and parallel computing. The closest work we know of is the work on parallel checkers described in Rubinfeld's thesis [Rub90]; however, that work uses the PRAM model of computation while we use a network model. Clearly, checking/correcting issues are related to self-stabilization, since in order to return a (distributed) program to a legitimate state, one must first detect an error, and then correct it.

One approach to check a distributed program [KP90] is to collect all information at a single "leader" node, thus reducing distributed checking to centralized checking. However, the best existing self-stabilizing leader-election algorithm [AKY90] requires time quadratic in the network diameter. Moreover, the space and processing overhead at the "leader" node grows with the size of network.

We will see in Section 2.3 that we can improve performance by doing distributed program checking. The major difficulty is to find self-stabilizing implementations of this paradigm. Both our implementations (Sections 4 and 5) are in the form of *compilers* that transform a synchronous protocol into a self-stabilizing version for dynamic networks. In essence, we efficiently transform a solution for the most restrictive model (synchronous, static fault-free networks) to a solution that works in a very permissive model (dynamic networks with catastrophic faults).

Model and Notation The communication network is modeled as a graph G of node and channel processes. Because we wish our protocols to work in dynamic networks where the graph topology can change, the notion of an unchanging graph is misleading. Thus it is better to think of G as the network after all failures stop. Let D be an upper bound on the diameter, E the number of edges, and V the number of nodes in G.

Each network link delivers every message sent in FIFO order to the the receiver. In [APV] it is shown how to implement such a FIFO link in a self-stabilizing fashion; unlike that paper we do not assume that the channel can store at most one message. However, we do assume the ability to do a snapshot and reset over each link. These can be done in a self-stabilizing fashion for instance by employing the solutions of [KP90] over a single link.

A network protocol is a set of node programs, one for each node. A global state of the protocol is the state of all nodes as well the messages on links. For randomized protocols, any supposedly random bits in the initial state of a node can be arbitrarily corrupted and hence non-random. However, subsequent coin-tosses will produce truly random bits.

Non-interactive protocols form an important subclass of distributed algorithms. These are protocols whose correctness can specified by a relation (the I/O relation) between its input and output. For example, if the protocol has to compute a spanning tree then the output (the tree) should be a spanning tree of the input (the graph G). [2]

We say that a non-interactive distributed algorithm is *self-stabilizing* if, when begun in an arbitrary initial global state with input I, the protocol produces (after finite time) an output O such that (I, O) belongs to the input output relation. In this paper we concentrate on non-interactive algorithms.

Let π be an arbitrary (non-stabilizing) synchronous protocol with time complexity T_π and space complexity S_π. We introduce the notion of a checker. Suppose that a synchronous protocol χ is a checker for π. If χ is given as input the input of π as well as what purports to be the output of π, χ must detect (at some node) if the purported output of π is incorrect. Any deterministic protocol π has a trivial checker that simply runs π again.

2 Results

2.1 Complexity Measures

In any non-trivial network protocol there must be some node v whose output depends on the input at some other node. If the protocol is self-stabilizing, it must have executions in which messages are sent infinitely often. If not, we could initialize the protocol such that

[2] By contrast, mutual exclusion is an interactive task because its correctness must be expressed in terms of sets of valid behaviors.

no node is in a state in which message sending is enabled and v has the wrong output: the system will remain in this deadlocked, incorrect state. Thus, even if the protocol is non-interactive and has computed its output, it must continue to send messages if it is to be self-stabilizing. Thus we introduce a new complexity measure called *stabilization bandwidth*.

Our measures are:

Stabilization Time: Starting from an arbitrary initial global state, this is the worst case time the protocol takes to produce a correct output, assuming that messages are delivered within unit time, and that processing is infinitely fast.

Space: The worst case space used by a node program.

Stabilization Bandwidth: For this measure, we limit ourselves to self-stabilizing protocols that execute a distinct checking protocol χ even after reaching a legal state; the stabilization bandwidth is the worst case message complexity per link of the checking protocol. Clearly the checking protocol must be executed at least once every T time units – where T is the stabilization time – even after the protocol has stabilized. Hence this is really a bandwidth cost. For example, the stabilization bandwidth of the protocol in [KP90] is the worst case message complexity per link to do a snapshot, which is $\Omega(E+V)$.

2.2 Existing work

Self-stabilization was introduced by Dijkstra [Dij74]. Later work includes [BP89, GM90, KP90, DIM90, IJ90a] [IJ90b, AG90, AKY90]. While many fundamental problems have been tackled there is a lack of general methods. We know of only two general techniques:

- [KP90] shows how to stabilize distributed algorithms by doing centralized checking at a leader. Also [AKY90] described a self-stabilizing algorithm for leader election that took time quadratic in the number of network nodes. The combination of centralized checking and the need to elect a leader reduce the performance of the compiler; this is shown in the first row of Figure 1.

- [AKY90] suggests replacing global checking, by doing local checking of neighboring nodes followed by global correction; they apply their idea to the problem of constructing a spanning tree. [APV] takes the next natural step and shows how, in certain important cases, they can replace global correction by doing local correction of the state of a node and its neighbors. They apply their technique to some important *interactive* tasks such as end-to-end message delivery and network resets. By contrast, this paper concentrates on general techniques for *non-interactive* tasks, for many of which (e.g., Minimal Spanning Tree, Min Cost Flows, etc.) no locally-correctable implementation is known.

Method	Stab. Bandwidth	Stab. Time	Space
[KP90],[AKY90].	$E+V\cdot S_\pi$	$V+D^2+T_\pi$	$E+V\cdot S_\pi$
Rollback	$T_\pi\cdot S_\pi$	T_π	$T_\pi\cdot S_\pi$
Resynchronizer	$T_\pi\cdot S_\pi$	$T_\pi+D$	S_π
+ Fast checker	S_π	$T_\pi+D$	S_π

Figure 1: Comparison of the complexity of our compilers (*Resynchronizer* and *Rollback*) with existing ones.

2.3 Our results

In this paper, we ask the following question: *Is there a general technique to transform a large class of distributed algorithms into self-stabilizing versions that work on dynamic networks*? We answer this question in the affirmative by introducing a method of stabilizing most solvable *non-interactive* tasks. Our solutions are in the form of compilers that can compile synchronous protocols into self-stabilizing versions that have the same I/O relation when run on a dynamic network. The performance of our compilers is summarized in the last three rows of Figure 1

Our simplest compiler is the *Rollback* compiler that takes a *deterministic* protocol π as input and produces a self-stabilizing version of π. Our main contribution is a second compiler called a *Resynchronizer*. In its simplest form, the *Resynchronizer* takes a *deterministic* protocol π as input and produces a self-stabilizing version of π.

We can improve the performance of π in some cases if we can show that there is a fast checker for π. Let us call χ a fast checker if its time complexity $T_\chi = O(1)$ and its space is $O(S_\pi)$. If π has a fast checker χ , then the the *Resynchronizer* produces a self-stabilizing version of π that has the complexity measures summarized in the fourth row of Figure 1. We will see in Section 7.1 that a number of important tasks have checkers with $T_\chi = O(1)$. We will also see in Section 7.2 how to use the *Resynchronizer* to compile randomized, synchronous protocols.

Note that, when compared to the *Resynchronizer*, the *Rollback* construction removes the additive factor

of D in the stabilization time but increases the space by a factor of T_π. Thus *Rollback* is useful only for "fast" protocols that have $T_\pi \ll D$. The two compilers can be used to efficiently stabilize most known non-interactive tasks.

Some sample results obtained by applying the *Resynchronizer* are as follows. For the problems of computing a spanning tree and single source shortest paths we achieve $O(D)$ stabilization time and $O(\log V)$ for the bandwidth and space measures. This is in contrast to the previous best result ([AKY90]) that achieves $O(D^2)$ stabilization time and does no better in the other measures. For the problem of computing a maximum flow [Gol85, GT88] we achieve $O(V^2)$ stabilization time, which is as good as the time of the best non-stabilizing synchronous protocol.

The *Rollback* compiler gives good results when applied to symmetry breaking problems such as the problems of computing a Maximum Independent Set [AGLP89], $\Delta+1$ Coloring in sparse networks [GPS87], and Δ^2 Coloring in general networks [Lin87]. For instance, for $\Delta+1$ Coloring in sparse networks we achieve $\log^* V$ for all three measures. Clearly this is much better than the corresponding stabilizing protocol constructed using the [KP90] compiler. We will present a more comprehensive list of results in the final paper.

3 Distributed Checking

A deterministic sequential algorithm can make itself self-stabilizing by periodically saving its output and running itself again; when it finishes it can check its output. For sequential algorithms, this is ugly and unnatural – after all, we want the computer to move on and run other programs!

However, this paradigm is quite natural for distributed computing. Unlike in sequential computing the output of a distributed algorithm is often used for a long period, sometimes for a period of several days. This is especially true if the output is used by a number of other protocols. For instance, most commercial networks compute paths to route packets between nodes; these paths are used by all [3] upper level protocols. For the path computation protocol to be self-stabilizing it must incur periodic bandwidth and computation. Because path computation is crucial, many networks (e.g. [MRR80]) [4] often check these paths periodically. A small portion of the node processing and link bandwidth is reserved for the overhead of self-stabilization.

[3] For example, by file transfer and mail.

[4] While the systems community has recognized the need for self-stabilization in their protocols, their mechanisms are somewhat ad hoc and are not supported by proofs.

Once we have accepted the inevitable periodic cost of self-stabilization we can ask: *why not run a checking process to check the algorithm periodically?* If the check reveals a problem, we restart the main algorithm.

However, in a distributed system we have to coordinate all the network nodes. For example, we need to ensure that a node not move to a new phase (whether it be checking or executing) before every other node in the network has completed this phase. The main difficulty is to implement this coordination in a self-stabilizing fashion. We will describe two such implementations – the *Rollback* protocol in Section 4 and the *Resynchronizer* in Section 5.

4 Rollback Compiler

There is a naive implementation of distributed checking that requires a large amount of storage and bandwidth and works only for deterministic protocols. In the naive method, every node keeps a log of every state transition it has taken to reach its current state. If each node constantly sends its current log to all neighbors, every node can check and correct every transition it has made in the past. Since the inputs are always correct, eventually all transitions are corrected. This method works only because it is possible to check transitions in a self-stabilizing fashion.

Clearly, for an arbitrary asynchronous protocol the logs can grow very large. However, if the asynchronous protocol is simulating an underlying synchronous protocol π then the size of the log can be reduced to the time complexity T_π of π. This idea is implemented in the *dynamic synchronizer* of [AS88]. However, in [AS88] the logs are only used to avoid unnecessary recomputation after an input change, and hence are not periodically checked. By adding the periodic checking of logs, we obtain a *Modified* Dynamic Synchronizer that we call *Rollback*.

The disadvantage of the *Rollback* method is that it blows up the space utilization and periodic bandwidth by a factor of T_π. This is not a problem for protocols with small time complexity – i.e., those for which $T_\pi \ll D$, where D is the diameter. Thus the naive method leads to efficient solutions for such problems as coloring and MIS. However, for protocols in which $T_\pi \geq D$, *Rollback* is a poor choice.

5 Resynchronizer Compiler

In the previous section, we saw how *Rollback* did distributed checking by maintaining a log of its computation. Consider a deterministic protocol π. Clearly, we can avoid the need for a log if we could simply

re-execute π. However, this constant re-execution requires coordination among the network nodes which must be implemented in a self-stabilizing fashion. In general, *Resynchronizer* avoids a log by constantly re-executing a checker χ for π. Let us introduce the basic idea by assuming that π is deterministic and that π is its own checker. We return to separate checking later.

The *Resynchronizer* can be thought of as a self-stabilizing version of a *synchronizer* [Awe85]. Any synchronous protocol can be simulated asynchronously by using a pulse number at each network node. Let us call a node *synchronized* if its pulse number differs by at most 1 from any of its neighbors. In ordinary synchronizers, every node is initially synchronized by setting the pulse numbers of all nodes to 0. Thereafter, a node increments its pulse number from p to $p+1$ only after all its neighbors have reached pulse p, thereby maintaining synchronization. If each node executes the corresponding step of the synchronous protocol at pulse p just before incrementing to $p+1$, then the asynchronous protocol has the same I/O relation as the underlying synchronous protocol.

Since a self-stabilizing synchronizer cannot rely on correct initialization, we introduce a *Resynch* phase. This phase will periodically ensure that all nodes in the network are synchronized, after which we can run the synchronizer protocol. Finally, we have a *Termination_Detection* phase that ensures that each node has finished executing the synchronous protocol. We implement each phase using disjoint ranges of pulse numbers.

Let us denote by T_e the pulse number at which execution of the underlying synchronous protocol begins. Let us denote by $\texttt{Pulse}_u$ the pulse number at node u. When $\texttt{Pulse}_u \in [0, T_e - 1]$ we say that node u is in the *Resynch* phase. This phase consists of dummy pulses – i.e., there is no execution of the underlying synchronous protocol. The objective of this phase is as follows. Suppose that initially all nodes are started in a "sufficiently early part" [5] of the *Resynch* phase. Then in $O(D)$ time some node will exit the *Resynch* phase and all nodes will be synchronized. T_e is chosen to be large enough to allow this objective to be met. Roughly speaking, during this phase each node tries to return to node synchronization with its neighbors by lowering its pulse to catch up with slower neighbors.

When $\texttt{Pulse}_u \in [T_e, T_e + T_\pi]$ we say that node u is in the *Execute* phase. In the *Execute* phase node u simply executes the normal synchronizer algorithm described earlier. It also implements the underlying synchronous protocol, starting from initialization at pulse T_e followed by writing its output at pulse $T_e + T_\pi$. We denote by $Max = T_e + T_\pi + D$ the maximum value of $\texttt{Pulse}_u$. When $\texttt{Pulse}_u \in [T_e + T_\pi + 1, Max]$ we say that node u is in the *Termination_Detection* phase. Suppose that all nodes are synchronized when some node exits the *Execute* phase. Then the *Termination_Detection* phase ensures that every node has had a chance to correct its output before the pulse number of some node wraps around to 0. If the nodes are synchronized by this phase, a node need only wait D more dummy pulses to make sure that all other nodes have reached the *Termination_Detection* phase.

Once $\texttt{Pulse}_u$ reaches *Max*, node u wraps around to 0, thus potentially destroying synchronization. However, we can show that within $O(D)$ time after some node reaches pulse 0, all nodes are in a "sufficiently early" part of the *Resynch* phase. Hence we can rely on the *Resynch* phase to restore synchronization as before, and the cycle continues. Notice that all nodes constantly re-execute the underlying synchronous protocol.

The *Resynch* phase is the heart of the compiler. First, it must deal with arbitrary pulse numbers and arbitrary messages on links in the initial state. Second, it must cope with the fact that the pulse numbers are finite and hence have to wrap around. Recall that one of our motivations for studying self-stabilization was to make real network protocols more fault-tolerant. Any real counter implementation is bounded.

In our description and in the code below we only describe the simplest form of *Resynchronizer* which can be used to compile a deterministic protocol by re-executing it. It is easy to extend these ideas slightly to add a separate checking phase to the *Resynchronizer*.

5.1 Resynchronizer Code

We described how to reduce the problem of stabilizing the output of a synchronous protocol π to the problem of building a self-stabilizing synchronizer that constantly re-executes pulse numbers in the execute region. Thus when presenting the code, for simplicity we ignore the details of executing π; instead we concentrate on the major task of synchronizing pulse numbers. To actually execute π, we need to supplement the code as follows:

- Additional state variables are added at every node u to keep track of the state of π as well as the last two messages received by u from every neighbor. Any messages sent by the synchronous protocol are piggybacked on the corresponding PULSE message.
- Whenever $\texttt{Pulse}_u$ reaches p and p is in the *Execute* phase, the synchronous protocol π is ex-

[5] this is made precise in the proof outline of Theorem 6.1

ecuted at node u.

- Whenever $\texttt{Pulse}_u$ reaches $T_e + T_\pi$, node u corrects its output to be the output of π.

The protocol is formally presented below in Figures 2, 3, 4 and 5. It uses the mechanisms developed in the second dynamic synchronizer of [AS88]; however the code in that paper does not have to deal with self-stabilization and pulse numbers wrapping around.

Every node keeps the following variables. **Pulse** describes the number of the last pulse performed correctly. **Parent** points to a neighbor which caused the most recent drop in the pulse number; the set of these pointers forms a forest of directed trees, rooted at the nodes which observed input changes. Each such tree is called a *drop tree*.

Once a node reaches **Pulse** $= Max$, it drops its **Pulse** to 0, sets **Parent** to *nil*. Next, it executes procedure BROADCAST-DROP. In that procedure the node broadcasts a **DROP(Pulse)** message to each adjacent neighbor v , setting **Ack(v)** := *false*. Once a node receives a **DROP(p)** message from neighbor v, it acts as follows. If **Pulse** $\leq p+1$, then it sends back **ACK(p+1)** message. Otherwise, it sets **Pulse** $:= p+1$, and **Parent** $:= v$, and executes procedure BROADCAST-DROP.

The meaning of the **ACK** message is that the subtree of the drop tree rooted at that node has stopped growing. A node may send many **DROP** messages while it is dropping; however the pulse numbers in those messages are strictly decreasing. Thus, **ACK(p)** messages arriving at the node with $p >$ **Pulse** correspond to a previous **DROP** message. Once an acknowledgment **ACK(p)** arrives from neighbor v, the node acts as follows. If $p \neq$ **Pulse**, then this **ACK** is outdated and is discarded. Otherwise, if $p =$**Pulse**, this is the **ACK** to the latest **DROP** message sent by the node. In the latter case, the node sets **Ack(v)** := *true* and executes procedure CONVERGE.

In procedure CONVERGE, the node checks whether **Ack(v)** = *true* for each adjacent neighbor v. If so, the node will send **ACK(Pulse-1)** message to its **Parent**. Once the root of a drop tree detects that **Ack(v)** = *true* for all neighbors v, it resumes normal synchronizer operation.

For that purpose, it executes the procedure BROADCAST-GROW. In the procedure BROADCAST-GROW, the node sends a **GROW** message to all neighbors, and sets **Parent** := *nil*, thus resuming normal synchronizer operation.. Once a node receives a **GROW** message from its **Parent**, it executes procedure BROADCAST-GROW.

6 Local Correction for the *Resynchronizer*.

Clearly the code described in Figures 3 and 4 is not sufficient to guarantee self-stabilization. For instance, if in the initial state there are no messages in any channel the code will deadlock. Normally, such illegal states of a protocol are avoided by proper initialization. However, a self-stabilizing protocol cannot depend on initialization.

In place of initialization, the code uses an additional mechanism, the method of local checking and correction [APV] to force the code into legal states. Briefly the method is as follows. First, we describe the legal states of the *Resynchronizer* as a conjunction of local predicates. A local predicate $L_{u,v}$ is a predicate that refers only to variables in Node u, Node v, or to the channels between u and v. Once we list the local predicates $L_{u,v}$ for the *Resynchronizer* we can apply the method of local correction described in [APV]. The idea is that each node u periodically does a snapshot of the subsystem consisting of node u, node v, and the links between them. If the snapshot detects that $L_{u,v}$ is false the link subsystem is reset to force $L_{u,v}$ to be true.

Next we must show that each such predicate is independently stable – i.e., if $L_{u,v}$ is true in some state s of the *Resynchronizer* protocol (augmented with periodic local checking and correction), then $L_{u,v}$ is true for all states $\tilde{s}$ that can follow s. It is not hard to see that if the periodic checking is done every P units of time, and all predicates $L_{u,v}$ are independently stable then the *Resynchronizer* code will reach a state in which all its local predicates are true in $O(P)$ time. The reader is referred to [APV] for more details.

In order to apply the method of local checking we need to specify two things. First, we need to specify what procedure we invoke when we reset an edge at a node. Secondly, we need to specify the predicates $L_{u,v}$. To reset an edge at a node we simply execute the procedure RESET-EDGE shown in Figure 3.

Each $L_{u,v}$ can be written as a conjunction of a number of predicates that essentially describe the legal states of the (u, v) subsystem consisting of node u, node v and the channels between them. There is a simple way to generate this list. First start the protocol in Figures 3 and 4 in the following "legal" initial global state s. In s, for each pair of neighbors u, v: $\texttt{Growing}_u = true, \texttt{Pulse}_u = 1, \texttt{Pulse}_v(u) = 0$ and the only message in the channel from u to v is a **PULSE**(1) message. Now we write down predicates that describe the possible states of the (u, v) subsystem in all possible global states that succeed the global state s.

Unfortunately the complete list of local predicates is

quite long, and is deferred to the final paper. Thus for this abstract we only describe the predicates that are crucial to the proof of Theorem 6.1. These are shown in Figure 5. Note that we subscript code variables by a node name to describe a variable stored at a node; thus $\texttt{Pulse}_u$ denotes the value of variable `Pulse` at node u. Also when we use $\texttt{Pulse}_u(v)$ we assume that $v \in \texttt{Adjacent}_u$.

Using these predicates we can prove the following theorem:

Theorem 6.1 Suppose local predicates are checked every P units of time, $P = O(D)$, and [6] $T_e = 10D+50$. Consider any execution (beginning from an arbitrary global state) in which there are no topology changes. Then in $O(D + T_\pi)$ the output will be correct. As before, the output is correct if its belongs to the I/O relation of π on the final network.

Proof: The proof proceeds by proving the following claims:

- The local predicates (including those listed in Figure 5) are independently stable.
- Within time $T_1 = O(D + T_\pi)$ time after an execution of the protocol begins, some node u has $\texttt{Pulse}_u = 0$.
- Within time $T_2 = O(D) + T_1$ all nodes u have $\texttt{Pulse}_u \leq 5D + 30$.
- Within time $T_3 = T_2 + O(D)$ some node u has has $\texttt{Pulse}_u = T_e$ and all nodes are synchronized.
- Within time $T_4 = T_3 + O(T_\pi)$ the output of every node is correct.

7 Extensions

7.1 Better Synchronous Checkers for Deterministic Protocols

If we check a deterministic protocol π by re-executing π, we have to pay a high price (T_π) in stabilization bandwidth. Suppose instead, we can find a checker χ for π that has $T_\chi = 1$ – i.e., can check π in a single pulse. Then after the execution phase we can add a single check pulse number T_{cp}. When a node reaches pulse T_{cp} it stays at T_{cp} executing the checking protocol until it detects a problem; if it does detect a problem it drops back to pulse 0. By avoiding multiple pulses for checking, we remove the need for costly resynchronization and termination detection in the checking phase. The stabilization bandwidth drops to $O(S_\pi)$.

[6] The constants have been chosen to allow a simple proof; no attempt has been made to optimize the constants.

Variables

`Adjacent`: the set of adjacent neighbors; this is maintained by the Data Link.

`Pulse`: Highest pulse which was performed correctly until now. Its domain is restricted to lie in $\{1..Max\}$.

`Pulse(v)`: Estimate of `Pulse` of neighbor v.. Its domain is restricted to lie in $\{1..Max\}$.

`Ack(v)`: Binary flag, indicating whether an `ACK` is expected from neighbor v, maintained for all $v \in$ `Adjacent`.

`Parent`: The parent of a node in its current drop tree. Its domain is restricted to be a node in `Adjacent` or the special value *nil* which indicates that the node currently has no parent.

Figure 2: Declarations of variables used by *Resynchronizer* code.

There are a number of tasks that we can check in a single pulse. These include the classical problems of shortest paths, topology update, leader election, computing a spanning tree, and computing a maximum flow. Because the first four tasks are commonly used in real networks, it is important to improve their stabilization bandwidth.

We do so by using local checking. The idea of doing local checking was introduced in [AKY90] for a specific asynchronous spanning tree protocol. However, in our case all we have to do is to check a *synchronous* protocol. Hence our checking procedures are very simple.

The key is the ability to check, in a single pulse, the shortest cost distances from a given source to every other node. Let s denote the source and D_i denote the distance node i has recorded to s; let $B_{i,j}$ denote the cost of a link from Node i to Node j, and $N(i)$ the set of neighbors of Node i. Then in a checking pulse a) s checks that $D_s = 0$ b) Nodes other than s check that $D_i > 0$ and $D_i = Min_{j \in N(i)}(D_j + B_{i,j})$. It is quite simple to prove that if any of the distances are wrong some node will detect this after checking. By adding an extra distance label to the other tasks, the other tasks can also be checked in one pulse, using the same trick. We can do this, for instance, in leader election by adding the distance to the leader, and in maximum flow by adding the distance to the source in the residual graph.

In general, given our application, the design of faster checkers for synchronous protocols becomes an interesting and practical problem. For instance, can we check a minimum spanning tree in fewer pulses than it would take to compute the tree from scratch while us-

```
Predicates
  Dropping = true iff
    Parent ≠ nil and ∃v: Ack(v) = true
  Converging = true iff
    Parent ≠ nil and ∀v ∈ Adjacent: Ack(v) = true
  Growing = true iff
    Parent = nil and ∀v ∈ Adjacent: Ack(v) = true

Procedure NEW-PULSE
  while ∀v ∈ Adjacent, Pulse(v) > Pulse −1 do
    if Pulse < Max then
      Pulse := Pulse + 1
      for all v in Adjacent,
        send PULSE(Pulse) to v
    else Execute Procedure START-DROP

Procedure PARENT-SWITCH
  if Parent ≠ nil and Dropping then
    send ACK(Pulse-1) to Parent

Procedure START-DROP
  if Pulse ≠ 0 then
    Execute Procedure PARENT-SWITCH
    Parent := nil
    Pulse := 0
    Execute Procedure BROADCAST-DROP

Procedure RESET-EDGE(v)
  Pulse(v) := 0;
  Ack(v) := true
  if Pulse ≠ 0 then
    if Parent ≠ v Execute Procedure PARENT-SWITCH
    Pulse := 0
    for all j ∈ Adjacent, j ≠ v
      send DROP(Pulse) to j
      Ack(j) := false
  Parent := nil
  Execute CONVERGE

Procedure BROADCAST-DROP
  for all neighbors v ∈ Adjacent
    send DROP(Pulse) to v
    Ack(v) := false
  Execute CONVERGE

Procedure CONVERGE
  if for all v ∈ Adjacent, Ack(v) = true then
    if Parent ≠ nil then
      send ACK(Pulse+1) to Parent
    else Execute Procedure BROADCAST-GROW

Procedure BROADCAST-GROW
  for all v ∈ Adjacent
    send GROW to v
  Parent := nil
  Execute Procedure NEW-PULSE
```

Figure 3: Predicates and Supporting Procedures for *Resynchronizer* code.

```
For PULSE(p) from neighbor v
  Pulse(v) := p
  if growing then
    Execute Procedure NEW-PULSE

For DROP(p) from neighbor v
  Pulse(v) := p
  if p+1 ≥ Pulse then
    send ACK(p) to v
    if growing then
      Execute Procedure NEW-PULSE
  else
    Execute Procedure PARENT-SWITCH
    Pulse := p+1
    Parent := v
    Execute Procedure BROADCAST-DROP

For ACK(p) message from neighbor v
  if p = Pulse and Ack(v) := false then
    Ack(v) := true
    Execute Procedure CONVERGE

For GROW message from node v
  if v = Parent and Converging then
    Execute Procedure BROADCAST-GROW
```

Figure 4: Resynchronizer Code

ing only small storage? There are a number of similar open problems that arise from our work.

7.2 Randomized Protocols

We need a separate checker to compile a randomized protocol. This is because re-executing the original protocol can lead to a different output, and cause the checker to detect an error when there was none. Saving the original random bits in the state does not help either as these bits could be corrupted (see Model section). Further, this checker must be *oblivious*: it must not depend on the correctness of the supposedly random bits currently in the state. It appears that a self-stabilizing algorithm that uses a randomized checker needs an infinite supply of random bits since it cannot rely on the old random bits at any stage.

A simple example of compiling a randomized protocol is furnished by the problem of electing a leader in an anonymous network - i.e., a network in which nodes do not have any unique IDs. [7] Clearly we need randomization to break symmetry. To construct a self-stabilizing protocol for this task, we demonstrate a synchronous protocol for execution together with an

[7] If nodes have unique IDs we must assume that the IDs are protected; dropping this assumption makes the system more fault tolerant.

1. If $\texttt{Pulse}_u = x$ and $\texttt{Pulse}_v(u) < x$ then there is a $\texttt{DROP}(x)$ in the channel from u to v.
2. If there is a $\texttt{PULSE}(x)$ or $\texttt{DROP}(x)$ message in the channel from u to v, then $x \leq \texttt{Pulse}_v(u) + 2$.
3. If $\texttt{Dropping}_v$ and $Parent_v = u$ then $\texttt{Pulse}_u < \texttt{Pulse}_v$.
4. If ($\texttt{Dropping}_v$ and $Parent_v = u$) OR (there is a $\texttt{DROP}(x)$ message in transit from u to v) OR (there is an $\texttt{ACK}(y)$ message in transit from v to u) then $Ack_u(v) = false$.
5. Let $x = \texttt{Pulse}_u$. If $Ack_u(v) = false$ then: either ($\texttt{Dropping}_v$ and $Parent_v = u$) OR (there is a $\texttt{DROP}(x)$ message in transit from u to v) OR (there is an $\texttt{ACK}(x)$ message in transit from v to u).
6. Let $x = \texttt{Pulse}_u$. If $\texttt{Converging}_v$ and $Parent_v = u$ then: either ($\texttt{Growing}_u = false$ and $\texttt{Pulse}_u < \texttt{Pulse}_v$) OR (there is $\texttt{GROW}$ message in transit from u to v) OR (there is an $\texttt{ACK}(x)$ message from v to u).
7. If $\texttt{Pulse}_u(v) = Min_{j \in \texttt{Adjacent}} \texttt{Pulse}_u(j)$ and $\texttt{Growing}_u$ then $\texttt{Pulse}_u = \texttt{Pulse}_u(v) + 1$.
8. $\texttt{Pulse}_u \leq \texttt{Pulse}_u(v) + 1$.
9. If $\texttt{Growing}_u$ then $\texttt{Pulse}_u(v) \leq \texttt{Pulse}_u + 1$.

Figure 5: Local Predicates for *Resynchronizer* code that are crucial to the proof of Theorem 6.1.

oblivious synchronous checker.

In the execution protocol, each node picks a random ID uniformly and independently in a space of $1..X$. In the next D pulses, a node considers itself as the leader if it finds out that its ID is the largest in the network. In the checking protocol, each node i that considers itself a leader picks a new random value $\texttt{test}_i$ in the space $1..X$ and broadcasts $\texttt{test}_i$ during the next D pulses. At the end of D pulses, a node detects an error if it has received either no `test` values or it has received more than one `test` value. While both checking and execution can fail, by picking X to be a polynomial (of sufficiently high degree) in the number of nodes, we can ensure that a correct output will be produced in constant expected number of phases. Since each phase takes $O(D)$ the *expected* stabilization time is $O(D)$.

A more efficient protocol for this purpose (that works in time proportional to the actual diameter as opposed to a bound on the diameter) is given in [DIM91]. However, our solution seems to be simpler.

As in the case of deterministic protocols, checking randomized synchronous protocols seems an interesting research area. In general, the area of self-stabilization and distributed checking of synchronous protocols may provide a link between distributed computing and two other branches of theoretical computer science: parallel algorithms and sequential checking.

Summary and open problems

The main result of this paper, the *Resynchronizer*, is a compiler that transforms any synchronous protocol into a self-stabilizing version for dynamic asynchronous networks. The transformation adds $O(D)$ overhead to the time complexity of the protocol, where D is a bound on the diameter of the network after arbitrary failures. Clearly D can be much larger than the actual diameter of the final network. A natural open problem is to obtain a compiler whose time overhead only depends on the actual diameter of the final network.

Acknowledgments

We would like to thank Yishay Mansour for helping us find a bug in an early version of the protocol, and Boaz Patt for helpful comments that improved the presentation of this paper.

References

[AG90] Anish Arora and Mohamed Gouda. Distributed reset. In *Proc. 10th FSTTSC, pages 316-331, Springer Verlag (LNCS 472*, 1990.

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

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th International Workshop on Distributed Algorithms, (eds. J. van-Leeuwen and N. Santoro), Bari, Italy, September 1990, Lecture notes in Computer Science 486 Springer Verlag, pp. 15-28.*

[APV] Baruch Awerbuch, Boaz Patt, and George Varghese. Self-stabilization by Local Checking and Correction. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, 1991.

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

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

[Blu] M. Blum. Designing programs to check their work. Submitted to *Communications of the ACM.*

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

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

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

[DIM91] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election. Unpublished manuscript, 1991.

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

[GM90] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. Unpublished manuscript, 1990.

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

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

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

[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.

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

[MA89] Yossi Matias and Yehuda Afek. Simple and efficient election algorithms for anonymous networks. In *Proceedings of the 3rd International Workshop on Distributed Algorithms: Springer-Verlag LNCS 392*, pages 183–194, September 1989.

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

[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.

[Rub90] Ronitt Rubinfeld. A mathematical theory of self-checking, self-testing, and self-correcting programs. Technical Report TR-90-054, ICSI, October 1990.

[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.