Self-Stabilization By Local Checking and Correction

Extended Abstract

Baruch Awerbuch* Boaz Patt-Shamir† George Varghese‡

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

Abstract

In this work we introduce the first self-stabilizing end-to-end communication protocol, and the most efficient known self-stabilizing network reset protocol. We use a simple method of local checking and correction, by which distributed protocols can be made self-stabilizing without the use of unbounded counters.

1 Introduction

Since the concept of self-stabilization was conceived by Dijkstra [Dij74], it has attracted considerable attention in the distributed computation area. Informally, a protocol is said to be self-stabilizing if its specification does not require a certain "initial configuration" to be imposed on the system to ensure correct behavior of the protocol. Alternatively, a system is called self-stabilizing if there is a subset of its state set, called the set of "legitimate states", such that (i) every successor of a legitimate state is legitimate, and (ii) starting from any state, the system eventually reaches a legitimate state.

Self-stabilizing protocols are appealing in two ways. First, from a theoretical point of view, a self-stabilizing protocol is "cleaner", since the disturbing need to specify an initial state is eliminated. Secondly, allowing a protocol to be initialized in an arbitrary state can be viewed as an abstraction of a strong fault tolerance property, namely the ability to tolerate *any* transient error. In particular, a self stabilizing protocol can recover from memory corruption, a property that none of the benign failure models (e.g., fail-stop, omission) can handle. The self-stabilization model is especially appropriate for the case of "infrequent catastrophes": every once in a (long) while the system may crash, resulting in an arbitrary state. A self-stabilizing protocol guarantees that eventually the system will regain stability in some legitimate state. Hence, relevant complexity parameters of a self-stabilizing solution are stabilization time, and the overhead in time/communication when the system is operating in a "regular mode" (i.e., when all states are legitimate), as compared to a "conventional" protocol solving the same problem.

In the self-stabilization model we distinguish between *catastrophic faults* that abstract arbitrary corruption of global state, and other restricted kinds of *anticipated faults*. It is assumed that after the execution starts there are no further catastrophic faults, but the anticipated faults may continue to occur. In this paper we consider two models of anticipated faults: *dynamic networks*, in which links can crash and restart, but failure is detectable; and *fail-stop networks*, in which links may crash without notification.

In this work we present self-stabilizing protocols for two major interactive tasks in distributed computing: *reset* in dynamic networks, and *end-to-end communication* in fail-stop networks. Informally, the reset problem [Fin79] is to design a reset service that can be superimposed on any other distributed protocol. The reset may be invoked at any node, and its effect is to output restart signal at all the nodes of

*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 ONR contract N00014-85-K-0168, by NSF grants CCR-8915206, and by DARPA contracts N00014-89-J-1988.

‡Supported by DEC Graduate Education Program.

the system in a consistent way. The reset protocol has to deal with links crashing and coming up, but link failure is detectable. The end-to-end communication [AG88] is roughly described as follows: deliver data items generated at a designated sender node to a designated receiver node across an unreliable network, without duplication, loss, or reordering. The network topology is fixed, but links may fail forever without notification.

To solve these two problems, we introduce a method of converting a distributed protocol into a self-stabilizing one. By applying this method, we obtain the first self-stabilizing end-to-end communication protocol, and the best known self-stabilizing reset protocol.

Existing non-stabilizing solutions. In 1979, Finn [Fin79] introduced the notion of network reset as a general paradigm to automatically extend a fixed-network protocol to work in a dynamic network. In Finn's solution, the nodes run the fixed-network protocol as long as the network does not appear to change. Whenever a node detects a change of the topology, it initiates a new version of the protocol. Finn's method to distinguish between old and new versions is to append "instance identifier" (i.e., a running counter) to the messages. The number of bits in the counter is large enough to avoid wrap-around.

As in the case of network reset, the main obstacle in the end-to-end communication is how to distinguish between old and new packets, in order to discard the former and forward the latter. This can be easily accomplished by maintaining a counter at the source, incrementing it for each new message sent, and stamping each message with the counter value. In this method each packet traverses each link at most once; hence its complexity appears to be linear. However, using appropriately defined measures, the complexity of this solution turns out to be unbounded. This is because the counter value grows indefinitely and cannot be bounded by a function of the network size.

In [AAG87, Awe88] more complex "bounded-counter" network reset protocols have been developed. Bounded-counter end-to-end protocols have been developed in [AG88, AMS89, AGR91, AG91]. None of them is self-stabilizing. This issue is addressed by Lamport and Lynch in their survey of distributed computing [LL90]:

> "...simply bounding the number of instance identifiers is of little practical significance, since practical bounds on an unbounded number of identifiers are easy to find. For example, with 64-bit identifiers, a system that chooses ten per second and was started at the beginning of the universe would not run out of identifiers for several billion more years. However, through a transient error, a node might choose too large an identifier, causing the system to run out of identifiers billions of years too soon — perhaps within a few seconds. A self-stabilizing algorithm using a finite number of identifiers would be quite useful, but we know of no such algorithm."

Related self-stabilizing protocols. Recently, an extensive effort was directed toward finding efficient self-stabilizing solutions for some of the basic tasks of distributed computing. Afek and Brown [AB89] give a self-stabilizing protocol for a perfect FIFO channel built on top of physical channels that may lose messages. Their construction utilizes an aperiodic (or random) number generator. This is no coincidence: an impossibility result by Dolev, Israeli and Moran [DIM91] shows that no self-stabilizing deterministic finite-state machine can cope with physical channel that is allowed to lose (and not even to permute) messages, if the channel capacity is unbounded. However, if a bound on the physical channel capacity is known, a finite-state self-stabilizing protocol is feasible.

Arora and Gouda [AG90] present a self-stabilizing reset protocol, using a self-stabilizing spanning tree of the network. For the spanning tree protocol, it is assumed that processes have unique identifiers, and that there is some *a prioi* bound K on the number of nodes in the network. The IDs and K cannot be corrupted by transient errors. Instance identifiers are used to distinguish between old and new messages. The stabilization time of this protocol is $O(K^3)$, where K is the non-volatile bound on the number of nodes in the network.

In the innovative works by Katz and Perry [KP90], and by Afek, Kutten and Yung [AKY90], general schemes to extend distributed protocols to be self-stabilizing are proposed. In [KP90], a fixed leader periodically takes snapshots (see [CL85]) of the system and checks them. If a "bad" global state is detected, a pre-determined global state is imposed on the system by the snapshot mechanism. In order not to confuse old snapshots with fresh ones, a global counter is maintained by the leader. Informally, this approach can be summarized as global (i.e., centralized) checking and global correction. In [AKY90], a new idea is introduced. They suggest that the protocol should maintain sufficient information at the nodes, so that illegal states can be detected *locally* by some node that can restart the algorithm. This approach can be viewed as local checking with global correction (the algorithm needs to be restarted in order to reach a legal state). They use this idea in a self-stabilizing spanning tree protocol that does not require counters. They propose to implement a reset service using it, and to combine it with the protocol of [KP90]. The stabilization time of their protocol is $O(V^2)$.

Results and organization of this paper. In this work we take the next natural step: we focus on protocols that are locally checkable, and locally *correctable*. Specifically, a protocol is said to be locally correctable if the global state of the protocol can be corrected to a legitimate global state merely by applying independent local actions, thereby exhibiting fast stabilization.

In Section 2 we describe a method to make protocols self-stabilizing by periodic local checking and correction. To implement such a mechanism, we assume that we are given *data links that can store at most one outstanding packet at a time*. On one hand, since in the so-called "real world" the capacity of communication channels is always bounded, such a data link protocol is realistic. On the other hand, dealing with unbounded capacity is doomed to result in infeasible infinite state machines [DIM91]. Therefore we argue that this assumption is reasonable, and moreover, by virtue of the results accomplished using this viewpoint, it seems fruitful for practical applications.

We justify this assumption in Section 3, with explicit constructions of a self-stabilizing unit capacity data links (UCDL). For physical channels whose capacity is bounded and may only lose packets, the protocol of Afek and Brown [AB89] yields UCDL. Adapting the strategy of [AGR91], we show how to implement a self-stabilizing UCDL on top of a dynamic network. Specifically, this is done in two stages. First, using the method introduced in Section 2, we show how to convert a dynamic network into a self-stabilizing bounded capacity channel, i.e., a channel whose capacity is bounded, and which may permute the packets. We then show how to implement a self-stabilizing UCDL over such a channel. Combined together, these protocols constitute the first self-stabilizing end-to-end communication protocol for dynamic networks.

In Section 4, we apply our method to the important task of network reset. Based on [AAG87], we present a self-stabilizing reset protocol with $O(1)$ stabilization time. The protocol does not require unique IDs, and its space overhead is logarithmic. The termination time of the reset protocol of [AAG87] is $O(V)$. Note that this protocol provides a powerful tool that can be used by existing protocols to make them self-stabilizing. For instance, when combined with any linear time static spanning tree algorithm, it yields $O(V)$ self-stabilizing spanning tree construction.

The model. Our model of computation is a variant of the Input/Output Automata model, described by Lynch and Tuttle [LT89].[1] For the time analysis purposes we make the standard assumption that with each action there is a certain time associated, such that a message is delivered within one time unit. We assume that executions start at time 0.

We recall that a *behavior* of an automaton A is the set of sequences of external (input/output) actions generated by all the executions of A. A *problem* is a set of behaviors; an automaton is said to *solve* a problem Π if its set of behaviors is a subset of Π. Usually, there is a set of *initial states* specified for an automaton, and we consider only executions (and behaviors generated by executions) starting in some initial state. For a self-stabilizing automaton, however, the situation is different. We use the following definition, due to N. Lynch [Lyn91]. An IO automaton A is a *self-stabilizing* automaton solving problem Π, if for any fair behavior β of A (regardless of the initial state), there exists a behavior $\gamma \in \Pi$, such that there is a sequence α which is a suffix of both β and γ. Intuitively, this definition says that an automaton is self-stabilizing if after some finite time, it behaves correctly. The *stabilization time* of a behavior is t if after time t, the behavior is a suffix of a correct behavior. The stabilization time of an automaton is the worst-case stabilization time of all its fair behaviors.

2 Local checking and correction

In this section we describe a method for the transformation of some distributed protocols into self-stabilizing protocols solving the same problem.

We begin with some standard definitions. Define the *global state* of a network protocol to be the Cartesian product of the local states of the nodes and the channels. Define *legal global states* to be the states, starting from which, the external behavior of the protocol is a suffix of a correct behavior. Note that once the protocol reaches a legal state, we can say that the protocol has stabilized, in the sense that the global state will remain legal.

We proceed by defining the notion of local checkability. For an edge $e = (u, v)$, the *e subsystem* is the 4-tuple that consists of the nodes u and v, and the two anti-symmetric links connecting them. For

[1] In the IO Automata model, the input events are always enabled. In order to avoid causality loops when composing automata, an action is not allowed to be both input and output action. For ease of presentation, we sometimes violate this restriction, but it should be clear that no causality loops arise.

a state s, denote by $[s]_e$ the projection of s to the e subsystem.

Definition 2.1 Let e be an edge. A predicate L is a *link predicate* for e if $[s]_e = [\tilde{s}]_e$ implies $L(s) = L(\tilde{s})$ for any two global states s and $\tilde{s}$.

We say that a network protocol is *locally checkable* if its set of legal states can be expressed as a conjunction of link predicates.

Some protocols may be *locally correctable*, i.e., even if each subsystem is corrected independently, the global state will eventually be legal. In the rest of this section we identify simple properties that imply "local correctability".

One straightforward condition for a locally checkable protocol to be locally correctable, is that every link predicate is stable: a stable predicate remains true regardless of whether other link predicates are true. We generalize the notion of stability to *dependent stability* in the following definition.

Definition 2.2 Let $L_1, \ldots, L_k$ and L be link predicates. L is *stable depending on* $L_1, \ldots, L_k$, if $L \wedge \bigwedge_{i=1}^{k} L_i$ hold in a state s implies that L holds in in all possible successor states of s.

It is not hard to see that if this dependency relation is acyclic then the protocol is locally correctable. In other words, if each link subsystem is independently corrected, then the entire system will stabilize in a time proportional to the length of a maximum dependency chain.

We now turn to the final obstacle, namely how to locally check and correct each link subsystem. Our strategy is to have each node periodically take snapshots [CL85] of each of its incident links, and verify whether the corresponding link predicate holds. If it does not hold, the node *resets* the link subsystem to a state in which the link predicate holds.

It may be interesting to compare this method with [KP90] and [AKY90]. In [KP90], the strategy is to take *global* snapshots and to reset the whole network if an illegal state is detected. In [AKY90], the idea is local detection of illegal states, complemented with resetting the algorithm. Our method is limited to locally checkable and correctable protocols, but by doing so we manage to avoid the inefficiencies incurred by the need to make global operations.

It is worthwhile pointing out that self-stabilizing snapshot and reset of links is easy to implement, especially when we assume that the links may contain at most one outstanding message at a time (e.g., use [KP90]). This assumption is justified in the next section.

3 Unit capacity data links

In this section we consider the model of a unit capacity data link (UCDL). We give a precise definition of UCDL in the IOA language, and sketch how is it implemented on top of physical channels by the protocol of [AB89]. Then, in the main part of this section, we follow the approach of [AGR91], and give a protocol for self-stabiling UCDL over a fail-stop network in two parts. First, by applying the method outlined in Section 2 to the protocol of [AGR91], we show how to convert a fail-stop network into a bounded capacity channel that may permute the packets but not to lose them. (We remark that the method can also be applied to the protocol of [AMS89], to yield a bounded capacity channel.) We complete the construction with a protocol that implements a unit capacity data link over such a channel, thus effectively exhibiting a self-stabilizing end-to-end communication protocol.

Throughout, we use a standard terminology, and call *packets* the internal information units exchanged over the channel, as opposed to *messages*, which are the information units that are transferred over the data link.

We commence with the specification of unit capacity data links. Let Σ be a fixed message alphabet. The external actions of a UCDL are *Free* (output action), and for each $m \in \Sigma$, $Send(m)$ (input) and $Receive(m)$ (output). A UCDL has $|\Sigma| + 1$ states: a distinguished *Free* state, and $|\Sigma|$ other states (*Busy* states), each corresponding to a distinct symbol. Intuitively, the states describe the contents of the link. In detail, the *Free* state corresponds to an "empty" link, in which the *Free* action is enabled; when a $Send(m)$ occurs, the state changes to m. In a *Busy* state m, $Receive(m)$ is enabled, and *Free* is disabled. *Send* events have no effect on a *Busy* state. When a $Receive(m)$ is taken, the state changes back to *Free*.

It is sometimes convenient to describe the behaviors of links as strings. Using regular expression operators, a behavior of UCDL can be broken to segments of the form

$$Free^*\ Send(m)\ Send(\_)^*\ Receive(m)\ ,$$

where $m \in \Sigma$, and "_" denotes an arbitrary symbol of Σ.

A self-stabilizing UCDL attains this behavior after some "stabilization time" has elapsed.

We now turn from abstraction to reality. A physical channel typically has finite capacity, and can only lose (but not permute) packets. Afek and Brown [AB89] use two such two anti-symmetric channels to construct a UCDL. Basically, their scheme is that the sender chooses a label for each message, and repeatedly transmits the message stamped by the current label, to the receiver, until the receiver acknowledges the current label. The link's state is *Free* from the time an acknowledgement is received until the next *Send* event is input. By choosing a label sequence with periodicity greater than the channel capacity, this protocol is a self-stabilizing UCDL. The stabilization time is proportional to the capacity of the channel.

However, there are cases in which we want to convert a network into a data link. Unfortunately, networks have the ability to store and permute packets. In that case the method of [AB89] does not apply.

We consider the following setting, called the end-to-end communication [AG88]. The network topology is fixed, but links may crash forever without notification (fail-stop network). There are two distinguished nodes, called *sender* and *receiver*. The task is to deliver a sequence of messages from the sender to the receiver without omission, duplication or reordering. It is assumed that there is no permanent cut of failed links between the sender and the receiver. In the remainder of this section, we solve a stricter problem, namely converting a fail-stop network into a UCDL.

Following [AGR91], our solution consists of two parts. We define an intermediate specification, the *bounded capacity channel*. We first show how to convert a fail-stop network into a self-stabilizing bounded capacity channel, and then present a protocol that converts such a channel into a self-stabilizing UCDL.

We start with a generalization of the UCDL, introduced in [AGR91], the C*-channel*. The external actions of a C-channel are same as of a UCDL, namely *Free*, *Send* and *Receive*. Intuitively, a C-channel can store up to C packets and may deliver any of the packets currently stored. Each state is associated with a multiset of elements of Σ whose size is at most C. *Send*(m) changes the state by adding m to the state-multiset, if the size state-multiset is less than C. *Receive*(m) can be taken only if m is in the state-multiset, and its effect is to subtract m from it. In addition, a C-channel must guarantee that if no *Send* actions are input, then eventually *Free* is taken, and that in any execution fragment in which $C+1$ *Send* actions occurred in a *Free* state, at least one *Receive* occurs.

We now show how to convert a fail-stop network into a C-channel. Our strategy is to apply the method of local correction to the end-to-end protocols of [AMS89, AGR91]. These protocols, when correctly initialized, convert a fail-stop network into a C-channel. We replace the need for initialization by periodic local checking and correction.

Roughly speaking, in the protocols in [AMS89, AGR91] each node maintains "piles" of bounded size, in which arriving messages are stored. The pile of the designated sender serves as the input queue; new packets can enter the network only when this pile is not full. Packets entering the designated receiver's piles are immediately output.

Here, we concentrate on the solution that is simpler to prove, the Slide protocol [AGR91]. Informally, the Slide protocol works as follows. Nodes other than the sender and receiver have a pile of n *slots* associated with each incoming link; each slot has a unique *height* whose range is $[1..n]$. The sender has only one pile with one slot of height n, and at the receiver, all the piles have only one slot of height 1. The action of the algorithm is simple: each node continuously tries to remove the highest packets in its piles and send them to any slot of lesser height in any of its neighbors. When the sender's slot is empty, the *Free* action is enabled.

Let us examine closely some link $e=(u,v)$. Denote by `highest` the highest occupied slot at the e-pile at node v. In order that node u will know if it may send a packet over e, u maintains a local estimate of the maximal value of `highest`, called `bound`. In [AGR91] this is done by simple bookkeeping: `bound` is initialized to 1, and is incremented whenever u sends a data message to v; whenever v removes a message from its e-pile, it sends a "decrement" message to u.

Self-stabilizing code cannot depend on initial state. However, it turns out that the role of initialization can be summarized as follows. Let M denote the total number of data messages on transit from u to v, plus the number of "decrement" messages on transit from v to u. Then the initialization makes the following predicate true.

$$\texttt{bound} = \texttt{highest} + M + 1$$

It is easy to verify inductively that this predicate is stable, by considering the effect of all possible actions. Thus, assuming that the links are checked (and if necessary, corrected by setting `bound`) every $O(1)$ time units, we have a C-channel with $O(1)$ stabilization time and $C = O(VE)$ capacity.

The second part of the solution is implementing a self-stabilizing UCDL over a self-stabilizing C-channel. Let us first describe a simple variant of our protocol, that can be viewed as a combination of the majority scheme in [AGR91] and the alternating bit protocol [BSW69]. The sender and the receiver maintain bits, called `S_Bit` and `R_Bit`, respectively. The sender transmits each message in $3C+1$ copies, all tagged by `S_Bit`, then negates the bit and proceeds to the next message. The receiver maintains a list of the last $3C + 1$ packets received. When this list contains $2C + 1$ packets tagged by R_Bit, then a packet that occurs at least $C + 1$ times is output (stripped of its tag), and R_Bit is negated.

Note that replicating each message in $3C+1$ copies is a considerable overhead. In [AGR91], it is proposed to use Rabin's information dispersal algorithm [Rab89] to reduce this overhead for "long" messages. The idea is to send a large number of different packets in a batch, thus amortizing the overhead needed to eliminate the effect of old packets. We use a slightly different scheme. Specifically, the strategy is as follows. Each packet is tagged by its position in the batch. Notice that although old packets can be confused with correct ones, when we send a batch of B packets, the receiver (after reconstruction) is guaranteed to get at least $B - 3C$ correct packets *in correct position* (when reconstructing, we may have at most C collisions of positions, each of which can cause bad locationing of a correct packet). Hence, by applying to every batch an error-correcting code that tolerates up to $3C$ errors, we are done. If we use BCH coding [MS78], each packet will contain $O(\log C)$ bits, and the number of packets in a batch will be $B = O(C \log C)$. For messages whose size is $\Omega(C \log C)$, the amortized communication overhead of this scheme is a constant.

The generalized protocol is given in Figure 1. The full specification of the protocol depends on C and the messages size. If the size of the messages is $o(\log C)$, then the protocol is the simple variant described above: $B = 3C + 1$, $encode(m)$ produces a batch of $3C + 1$ copies of m, and $decode$, given a set of at least $2C+1$ packets, outputs the packet which occurs at least $C+1$ times (if there is no such packet output arbitrary value). In the second case, where the messages are "large", $B = O(C \log C)$; $encode(m)$ is computed by breaking m into subsequences of $O(\log C)$ bits, applying BCH encoding, and then tagging the packets by their position. Messages are reconstructed by $decode(P)$ as follows. First, the packets are positioned according to their position tags: in case of a conflict, one of the packets is located in the claimed position, and the others are located arbitrarily in the unclaimed positions; empty locations are filled by dummy packets. Finally, BCH decoding is applied to the ordered batch, yielding the original message.

The sender uses an array S_Packet_Q of fixed size B, and a counter Count that indicates the number of packets pending to be sent. We denote by $tag(q, b)$ a q tagged by a bit b, and for a packet p whose tag is b, we denote $bit(p) = b$. The sender also uses a bit Free_C, to indicate the status of the C-channel. The receiver uses the bit R_Bit, and a queue R_Packet_Q of size B to record the last B packets received from the C-channel. The operation $update(\texttt{R\_Packet\_Q}, p)$ updates R_Packet_Q by removing the oldest packet and inserting p. The "Free" input action (from the C-channel) is denoted *Free_packet*, and the "Free" output action (required by the specification of UCDL) is denoted *Free*, and is enabled whenever sending the previous batch over the C-channel is completed.

The protocol above features fast stabilization, as stated in the theorem below. The proof of the theorem is omitted.

Theorem 3.1 The protocol in Figure 1 is a self-stabilizing UCDL. If the C-channel accepts B packets in T time units, then the protocol stabilizes in $O(T)$ time units.

We remark that this result is complementary to the elegant construction of [AB89], which cannot tolerate reordering of the packets.

```
Sender Protocol

Whenever Send(m)
  if Count = 0 then
    S_Bit ← ¬S_Bit
    S_Packet_Q ← encode(m)
    Count ← B

Whenever Count > 0 and Free_C = TRUE
  Send_packet(tag(S_Packet_Q[Count], S_Bit))
  Count ← Count − 1
  Free_C ← FALSE

Whenever Count = 0
  output Free

Whenever Free_packet
  Free_C ← TRUE

Receiver Protocol

Whenever Receive_packet(p)
  update(R_Packet_Q, p)
  P' ← {p' ∈ R_Packet_Q : bit(p') = R_Bit}
  if |P'| ≥ B − C then
    output Receive(decode(P'))
    R_Bit ← ¬R_Bit
```

Figure 1: Message delivery over a C-channel

4 Network reset

In this section we define the reset problem, and exhibit a self-stabilizing protocol that solves it. The protocol is based on the protocol of [AAG87], augmented by the method of local checking and correction presented in Section 2. In contrast to other solutions, this protocol does not construct (or assume the existence of) a self-stabilizing spanning tree, and does not require the processes to have unique IDs. Instead, the requesting node is used as the root of an *ad hoc* tree that disappears when the reset signal is output, and it is assumed that a process is able to distinguish among its incident links.

The *reset problem* is defined as follows. We are given a dynamic network, with a so-called user per each node. We'll sometimes identify the user with its node. The user has input action $Receive(m, e)$ (meaning "m is received from e"), and output action $Send(m, e)$ ("send m over e"), where m is drawn from some message alphabet Σ, and e is an incident link. The user also has output action *reset request* and input action *reset signal*. The problem is to design a protocol ("reset service"), such that after the topological changes stop, if one of the nodes makes a reset request and no node makes infinitely many requests, then (i) in finite time all the nodes in the connected component of a requesting node receive a reset signal, (ii) no node receives infinitely many reset signals, and (iii) if $e = (u, v)$ is a link in the final topology, then the sequence of $Send(m, e)$ input at u after the last reset signal at u, is identical to the sequence of $Receive(m, e)$ output at v after the last reset signal at v.

Intuitively, (i) and (ii) guarantee that every node gets a *last* reset signal, and (iii) stipulates that the last reset signal provides a consistent reference time-point for the nodes.

In order that the consistency condition (iii) will be satisfiable, it is assumed that all Σ-messages (from the user to the network and vice-versa) are controlled (i.e., buffered) by the reset protocol.

We consider the problem in a fail-safe network, i.e., we assume that the link continuously informs the nodes at both ends its operational state, by maintaining a dedicated bit. Whenever a change in the state of the link occurs (either "crash" or "recovery"), the dedicated bit is updated and the appropriate action is taken. We remark that although we model link failure or recovery as a simultaneous action observed at its endpoints, this simplifying assumption is not essential for the correctness of our protocol.

Briefly, the reset protocol of [AAG87] works as follows. In the so-called *Ready* mode, the protocol relays, using buffers, Σ-messages between the network and the user. When a reset request is made at some node, its mode is changed to *Abort*, it broadcasts ABORT messages to all its neighbors, sets the `Ack_Pend` bits to TRUE for all operational links, and waits until all the neighboring nodes send back ACK. If a node receives ABORT message while in *Ready* mode, it marks the link from which the message arrived as its `Parent`, broadcasts ABORT, and waits for ACKs to be received from all its neighbors. If ABORT message is received by a node not in a *Ready* mode, ACK is sent back immediately. When a node receives ACK it sets the corresponding `Ack_Pend` bit to FALSE. The action of a node when it receives the last anticipated ACK depends on the value of its `Parent`. If `Parent` $\neq$ NIL, its mode is changed to *Converge*, and ACK is sent on `Parent`. If `Parent` $=$ NIL, the mode is changed to *Ready*, the node broadcasts READY messages, and outputs a reset signal. A node that gets a READY message on its `Parent` link while in *Converge* mode, changes its mode to *Ready*, makes its `Parent` NIL, outputs a reset signal, and broadcasts READY.

While the mode is not *Ready*, Σ-messages input by the user are discarded, and all Σ-messages from a link e are queued on `buffer`$[e]$. When ABORT arrives on link e, `buffer`$[e]$ is flushed. While the mode is *Ready*, Σ-messages from the user to the network are put on the output queue, and Σ-messages in `buffer` are forwarded to the user. The size of `buffer` depends on the assumptions we make on the user. Here, we assume that at most one Σ-message is sent without getting a response, hence the size of `buffer` is 2.

We remark that the mode of a node is characterized by the state of the other variables as follows.

$$mode(u) = \begin{cases} Abort, & \text{if } \exists e \text{ such that} \\ & \qquad \texttt{Ack\_Pend}[e] = \text{TRUE} \\ Converge, & \text{if } \texttt{Parent} \neq \text{NIL} \textbf{ and} \\ & \forall e\,(\texttt{Ack\_Pend}[e] = \text{FALSE}) \\ Ready, & \text{if } \texttt{Parent} = \text{NIL} \textbf{ and} \\ & \forall e\,(\texttt{Ack\_Pend}[e] = \text{FALSE}) \end{cases}$$

The first step in making this protocol self-stabilizing is to make it locally checkable. A clear problem with the existing protocol is that it will deadlock if in the initial state some `Parent` edges form a cycle. As in self-stabilizing spanning tree algorithms [AKY90, AG90], we mend this flaw by maintaining `Distance` variable at each node, such that a node's `Distance` is one greater than that of its `Parent`. Specifically, `Distance` is initialized to 0 upon reset request, and its accumulated value is appended to the ABORT messages. However, since all we care about is acyclicity, there is no need to update `Distance` when a link goes down.

Next, we list all the link predicates that are necessary to ensure correct operation of the Reset protocol. Note that a significant advantage of our approach is that all we have to do is to prove that all link predicates eventually hold. Once we do that we can rely on the correctness of the original protocol. It turns out (perhaps surprisingly) that all the required link predicates are independently stable.

Our last step is to specify how to locally correct links when a violation of the predicates is detected. The main difficulty about designing a correcting strategy is making it local, i.e., to ensure that when we correct a link we do not affect the correctness of other links. In the case of dynamic network the solution is simple: we emulate a link failure. Since the original protocol had to deal with link failures anyway, it is clear that this method of correction does not invalidate other links' predicates. We observe that this process of deliberately failing the links terminates, since at most one such "correction failure" is generated per link, by the fact that the predicates are independently stable.

In order to be able to describe the protocol formally, let us agree upon some notational conventions. A subscript of a variable denotes at which node it resides. The first and the last messages in a buffer B are referred to as $head(B)$ and $tail(B)$, respectively. For a node incident to d links, we assume that the links are numbered 1 through d. A link (u, v) numbered e at u, is assumed to be numbered $\overline{e}$ at v. The links are assumed to be UCDLs, and since a UCDL is not always free to accept new messages, there is a buffer of messages pending to be sent, called `Queue`. The contents of a link e may be viewed as the top element `Queue`$[e]$, for the snapshot purposes.

The input actions are UP and DOWN (topological changes), reset request, and $Send(m, e)$ (meaning "send m over e"). The output actions are reset signal and $Receive(m, e)$ ("m received from e").

The main code for a node is given in Figure 2. The labels indicate action names. The code uses shorthand specified in Figure 3. The module that deals with Σ-messages, and the periodical checking and correction is called "manager", and its code appears in Figure 4. The link invariants verified by the manager are listed in Figure 5.

The correctness of the Reset protocol is stated in the following theorem, whose proof is omitted.

Theorem 4.1 Let n be the number of nodes in the network. Suppose that the links are self stabilizing UCDLs with stabilization time C, and suppose that invariants are verified by the manager at least every P time units. Then any execution of the Reset proto-

<u>State</u>

`Ack_Pend`:	array of d bits
`Parent`:	pointer, ranging over $[1..d] \cup \{\text{NIL}\}$
`Distance`:	ranges over the non-negative integers
`Up`:	set of operational incident links (maintained by the links protocol)
`Queue`:	array of d link buffers, each of size 3

<u>Actions</u>

Whenever reset request **and** $mode(u) = Ready$
Q: $Propagate(\text{NIL}, 0)$

Whenever ABORT$(dist)$ on link e
A: **if** $mode(u) = Ready$ **then**
    $Propagate(e, dist)$
  else enqueue ACK in `Queue`$[e]$

Whenever ACK on link e **and** `Ack_Pend`$[e]$ = TRUE
  `Ack_Pend`$[e] \leftarrow$ FALSE
K1: **if** $mode(u) = Converge$ **then**
    enqueue ACK in `Queue[Parent]`
K2: **else if** $mode(u) = Ready$ **then**
    output reset signal
    for all $e' \in$ `Up` **do**
      enqueue READY in `Queue`$[e']$

Whenever READY on link e **and** `Parent` $= e$
R: **if** $mode(u) = Converge$ **then**
    `Parent` $\leftarrow$ NIL
    output reset signal
    for all edges $e' \in$ `Up` **do**
      enqueue READY in `Queue`$[e']$

Whenever DOWN on link e
  `Queue`$_u[e] \leftarrow \emptyset$
D1: **if** $\exists e'(e' \neq e$ **and** `Ack_Pend`$[e']$ = TRUE) **or**
      ($e \neq$ `Parent` **and** `Parent` $\neq$ NIL) **then**
    if `Parent` $= e$ **then**
      `Parent` $\leftarrow$ NIL
    if `Ack_Pend`$[e]$ = TRUE **then**
      `Ack_Pend`$[e] \leftarrow$ FALSE
      if $mode(u) = Converge$ **then**
        enqueue ACK in `Queue[Parent]`
D2: **else if** $mode(u) \neq Ready$
    `Ack_Pend`$[e] \leftarrow$ FALSE
    `Parent` $\leftarrow$ NIL
    output reset signal
    for all $e' \in$ `Up` **do**
      enqueue READY in `Queue`$[e']$

Figure 2: Reset Protocol. Main code for node u, incident to d links.

$\mathit{Propagate}(e, \mathit{dist}) \quad \equiv$
$\quad \mathtt{Parent} \leftarrow e$
$\quad \mathtt{Distance} \leftarrow \mathit{dist}$
$\quad$ **for** all edges $e' \in \mathtt{Up}$ **do**
$\qquad \mathtt{Ack\_Pend}[e'] \leftarrow \text{TRUE}$
$\qquad$ enqueue $\text{ABORT}(\mathtt{Distance} + 1)$ on $\mathtt{Queue}[e']$

Figure 3: Reset Protocol. Shorthand for the code.

Manager state

$\mathtt{Free\_L}$: array of d Boolean flags, indicating the state of the links.
$\mathtt{buffer}$: array of d buffers, each of size 2.

Manager actions

Whenever $\mathit{Send}(m, e)$
$\quad$ **if** $\mathit{mode}(u) = \mathit{Ready}$ **then**
$\qquad$ enqueue m in $\mathtt{Queue}[e]$

Whenever $\mathtt{Queue}[e] \neq \emptyset$ **and** $\mathtt{Free\_L}[e] = \text{TRUE}$
$\quad$ send $\mathit{head}(\mathtt{Queue}[e])$ over e
$\quad$ delete $\mathit{head}(\mathtt{Queue}[e])$
$\quad \mathtt{Free\_L}[e] \leftarrow \text{FALSE}$

Whenever Free from link e
$\quad \mathtt{Free\_L}[e] \leftarrow \text{TRUE}$

Whenever $m \in \Sigma$ received from link e
$\quad$ enqueue m in $\mathtt{buffer}[e]$

Whenever $\mathtt{buffer}[e] \neq \emptyset$ **and** $\mathit{mode}(u) = \mathit{Ready}$
$\quad$ output $\mathit{Receive}(\mathit{head}(\mathtt{buffer}[e]), e)$
$\quad$ delete $\mathit{head}(\mathtt{buffer}[e])$

Whenever ABORT or DOWN on e:
$\quad \mathtt{buffer}[e] \leftarrow \emptyset$

Whenever $\mathtt{Free\_L}[e] = \text{TRUE}$ for some link e:
$\quad$ initiate snapshot on link e
$\quad \mathtt{Free\_L}[e] \leftarrow \text{FALSE}$

Whenever "snapshot completed" received from link e:
$\quad$ **if** any of the invariants does not hold
$\qquad$ crash and recover link e

Figure 4: Reset Protocol. Manager code for a node incident to d links.

$\mathcal{A}$: $\mathtt{Ack\_Pend}_u[e] = \text{TRUE}$ iff
$\quad$ one of the following holds.
$\qquad \text{ABORT}(\mathtt{Distance}_u + 1) \in \mathtt{Queue}_u[e]$
$\qquad \mathit{mode}(v) = \mathit{Abort}$ **and** $\mathtt{Parent}_v = \overline{e}$
$\qquad \text{ACK} \in \mathtt{Queue}_v[\overline{e}]$

$\mathcal{B}$: At most one of the following hold.
$\qquad \text{ABORT}(\mathtt{Distance}_u + 1) \in \mathtt{Queue}_u[e]$
$\qquad \mathit{mode}(v) = \mathit{Abort}_v$ **and** $\mathtt{Parent}_v = \overline{e}$
$\qquad \text{ACK} \in \mathtt{Queue}_v[\overline{e}]$

$\mathcal{C}$: $\mathtt{Parent}_u = e$ implies
$\quad \mathit{mode}(u) = \mathit{Converge}$ iff
$\qquad$ one of the following holds.
$\qquad\quad \text{ACK} \in \mathtt{Queue}_u[e]$
$\qquad\quad \mathtt{Ack\_Pend}_v[\overline{e}] = \text{FALSE}$ **and** $\mathit{mode}(v) \neq \mathit{Ready}$
$\qquad\quad \text{READY} \in \mathtt{Queue}_v[\overline{e}]$

$\mathcal{D}$: $\mathtt{Parent}_u = e$ implies
$\quad$ at most one of the following holds.
$\qquad \text{ACK} \in \mathtt{Queue}_u[e]$
$\qquad \mathtt{Ack\_Pend}_v[\overline{e}] = \text{FALSE}$ **and** $\mathit{mode}(v) \neq \mathit{Ready}$
$\qquad \text{READY} \in \mathtt{Queue}_v[\overline{e}]$

$\mathcal{R}$: $\mathit{tail}(\mathtt{Queue}_u[e]) \in \{\text{READY}\} \cup \Sigma$ implies
$\quad \mathit{mode}(u) = \mathit{Ready}$

$\mathcal{P}$: $\mathtt{Parent}_u = e$ implies
$\quad$ one of the following holds.
$\qquad \mathtt{Distance}_u = \mathtt{Distance}_v + 1$
$\qquad \text{READY} \in \mathtt{Queue}_v[\overline{e}]$

$\mathcal{E}$: $e \notin \mathtt{Up}_u$ implies all the following.
$\quad \mathtt{Ack\_Pend}_u[e] = \text{FALSE}$
$\quad \mathtt{Parent}_u \neq e$
$\quad \mathtt{Queue}_u[e] = \emptyset$
$\quad \mathtt{buffer}_u[e] = \emptyset$

$\mathcal{Q}$: $\mathtt{Queue}[e]$ is a subsequence of the following.
$\quad \langle \text{ABORT}, \text{ACK} \rangle$
$\quad \langle \text{ACK}, \text{READY}, \text{ABORT} \rangle$
$\quad \langle \text{ACK}, \text{READY}, m \rangle \qquad m \in \Sigma$
$\quad \langle \text{READY}, m, m' \rangle \qquad m, m' \in \Sigma$

Figure 5: Reset Protocol. Link invariants for link $e = (u, v)$. We denote $\overline{e} = (v, u)$.

col, regardless of the initial state, satisfies the following properties.

1. In all states from time $C + P$ onwards, all the invariants hold for all the links.
2. If the number of reset requests is finite, then $C + P + 3n$ time units after the last reset request, $mode(u) = Ready$ for all nodes u.
3. If there are no topological changes after time $C+P$, and the last reset request occurs at time $t > C + P + 3n$, then by time $t+3n$, a reset signal is output at all the nodes of the connected component in which the last reset request was made.
4. Suppose no topological changes occurs after time $C + P$, and that the last reset signals occur at two adjacent nodes u and v after time $C+P+3n$. Then the sequence $Send(m, e)$ input by the user at u following the last reset signal at u, is identical to the sequence of $Receive(m, \bar{e})$ output to the user at v after the last reset signal at v.

Acknowledgments

The authors are grateful to Yehuda Afek, Peter Elias, Eli Gafni, Nancy Lynch, Yishay Mansour, Nir Shavit, Mark Tuttle and Fritz Vaandrager for discussions that were crucial to the success of this research.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th FOCS*, 1987.

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

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliable networks. In *Proc. 7th PODC*, pages 131–148,1988.

[AG90] A. Arora and M.G. Gouda. Distributed reset. In *Proc. 10th FSTTCS*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AG91] Y. Afek and E. Gafni. Bootstrap network resynchronization. In *Proc. 10th PODC*, 1991.

[AGR91] Y. Afek, E. Gafni, and A. Rosen. Slide - a technique for communication in unreliable networks. Unpublished manuscript, 1991.

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

[AMS89] B. Awerbuch, Y. Mansour, and N. Shavit. End-to-end communication with polynomial overhead. In *Proc. 30th FOCS*, 1989.

[Awe88] B. Awerbuch. On the effects of feedback in dynamic network protocols. In *Proc. 29th FOCS*, pages 231–245, October 1988.

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

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

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

[DIM91] S. Dolev, A. Israeli, and S. Moran. Resource Bounds for Self-Stabilizing Message driven protocols. In *Proc. 10th PODC*, 1991.

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

[KP90] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 9th PODC*, 1990.

[LL90] L. Lamport and N.A. Lynch. *Handbook on Theoretical Computer Science*, chapter 18: Distributed Computing, pages 1159–1199. North-Holland, 1990.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3), September 1989.

[Lyn91] N.A. Lynch. personal communication, 1991.

[MS78] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North Holland Pub. Co., Amsterdam, 1978.

[Rab89] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. of the ACM*, 36(3):335–348, 1989.