Bounding the Unbounded

EXTENDED ABSTRACT

Baruch Awerbuch* Lab. for Computer Science MIT

Boaz Patt-Shamir† Lab. for Computer Science MIT

George Varghese‡ Dept. of Computer Science Washington University

Abstract

Many important protocols in distributed computing have simple and elegant solutions if we allow the assumption of unbounded size registers. This assumption can be simulated in practice using sufficiently large but bounded registers; however the resulting protocols are extremely vulnerable to transient faults. In this paper we present a general methodology for the transformation of unbounded register protocols so that they can work with bounded registers in a self-stabilizing fashion. The applicability of our method is demonstrated with two examples: spanning tree computation and topology update.

1 Introduction

The ability to count without bound is a useful tool in distributed computing. This approach, traditionally called "the unbounded registers model" [12] is the basis of many important techniques, such as time-stamps [7] and "logical time" [11]. It is implicitly used in many communication and synchronization protocols. Several problems yield simple solutions when unbounded registers are permitted. However, the unbounded-registers approach suffers from the obvious drawback that in real life, there is always is a bound on the number of bits that can be allocated to a register. The standard answer to this problem is to use a "sufficiently large" (e.g., 64 bit) counter, which is practically unbounded!

This argument is sound, but it implicitly assumes that *the system is initialized properly*. If the registers start working at an unknown, arbitrary state, then it may be the case that they hit the physically imposed bound of their value range. But in some networks, such initialization is hard to achieve. Also, the effect of a faulty node can be catastrophic; see, for example, the fascinating story of the crash of the ARPANET [21]. It is highly desirable, more than ever, to have *self-stabilizing* systems [6] — that is, systems that start working correctly regardless of their initial state or any transient fault. When one requires self-stabilization, one must take a new hard look at the model of unbounded registers. At first glance, it appears that a protocol will break if a fault introduces a value which is at the maximum for the physical register.

A common approach in practice to overcome such problems is to rely on approximately synchronized clocks at the nodes [17]. Each message is tagged with a timestamp, so that old counter values are eventually discarded. This method suffers from two disadvantages. First, the time limit needs to allow for the worst case propagation delay in the largest possible network, and is therefore quite high. This slows down the stabilization even in cases where it could have been attained quickly. Second, the value of the global timeout depends on all the components of the network; this makes the resulting protocols non-modular: changing the speed of one link might require changing the code at all network nodes.

In this paper, we describe a method by which unbounded-registers protocols can be efficiently adapted for use in a self-stabilizing fashion. The stabilization time of the resulting protocol is proportional to the *actual* end-to-end delay in a network, and not to a worst-case bound on the delay, which is typically much larger. Moreover, our scheme does not depend on global timing assumptions; rather, it depends only on time bounds on message delivery times over a single link and on message processing times at a single node. We demonstrate the applicability of our method with two example network protocols, a

*Supported by Air Force Contract AFOSR F49620-92-J-0125, NSF contract 9225124-CCR, DARPA contract N00014-92-J-4033, and a special grant from IBM.

†Supported by Air Force Contract AFOSR F49620-92-J-0125, ONR contract N00014-91-J-1046, NSF contracts CCR-9114440 and 9225124, DARPA contracts N00014-92-J-1799 and N00014-92-J-4033, and a special grant from IBM.

‡Work done while at Lab for Computer Science, MIT.

spanning tree protocol and a topology update protocol. Such protocols are commonly found in many real networks.

This paper is organized as follows. In Section 2 we describe the general paradigm by which unbounded-registers protocols can be made self-stabilizing with only finite physical registers. In Section 3 we describe a spanning tree protocol in some detail, and in Section 4 we briefly sketch the main ideas behind a topology update protocol.

2 The General Paradigm

Our paradigm has two parts. First, we add sufficient information to the state of the processes so that a fault can be efficiently *detected*. Hitting the bound for some register is considered a *fault*. The next step is that whenever a fault (in the broad sense described above) is detected, the detecting process invokes a *reset* of the entire network. This imposes a predetermined initial state on the system (e.g., all counters set to 0, all buffers flushed etc.). When the reset completes, the system starts operating according to its original specification.

At first, resetting an entire network seems drastic and inefficient. However, this technique has been implemented in the DEC SRC's AN-1 network [16]. The AN-1 designers found that the protocol recovered from link failures very fast[20]. In traditional networks, the routing protocol only operates for a small fraction of the time; the remainder is devoted to forwarding data. During a reset, however, no data forwarding is done; all processing and bandwidth is devoted to routing. AN-1 showed that reset schemes work well for small sized nets; for larger networks, the same approach should work if the routing protocol is hierarchical [10] and each level is reset independently.

In our method, the key to efficient fault detection is a concept called *local checkability* which ensures that faults can be efficiently detected by only examining the states of neighboring nodes. In the simplest case, this can be done by periodically sending the state of each node to its neighbors, which can then check for violations of local predicates. The case of a counter reaching the maximum value is also considered to be a fault. The key to efficient correction is a tool called a *self-stabilizing network reset protocol* which can be invoked by any node (if it detects a fault locally), and be used to restore the network to a good state. The basis for our method is a theorem that states that all locally checkable protocols can be stabilized using a network reset protocol. We cannot present the theorem here for lack of space (see [25] for details). Instead we concentrate here on applications and only specify the properties required from a reset protocol.

2.1 Local Checking

The network topology is modeled by a symmetric, directed graph $G = (V, E)$ in which node identifiers (e.g., $u, v \in V$) are unique. (G is symmetric if $(u, v) \in E$ iff $(v, u) \in E$.) Let $n = |V|$ denote the number of network nodes and D the network diameter. We model the nodes and links of the network using state machines called Input/Output Automata (IOA) [13, 15]. The network protocol is the composition of the node and link automata.

We now introduce the notion of local checkability. Roughly, a property is said to be local to a subgraph G' of the given network graph G if the truth of the property can be ascertained by examining only the components specified by G'. We will concentrate on *link subsystems* that consist of a pair of neighboring nodes u and v and the channels between them. It is possible to generalize our methods to arbitrary subsystems. Assume that we are given a network protocol $\mathcal{N}$. Formally, *the* (u, v) *link subsystem* of $\mathcal{N}$ consists of the nodes u, v, and the two directed links (u, v) and (v, u) connecting them. For any global state s of $\mathcal{N}$, we denote by $s|u$ the state of u in s and by $s|(u, v)$ the state of link (u, v). Thus when $\mathcal{N}$ is in state s, we can denote the state of the (u, v) subsystem using the 4-tuple $(s|u, s|(u, v), s|(v, u), s|v)$.

A *predicate* L of $\mathcal{N}$ is simply a subset of the states of $\mathcal{N}$. Let (u, v) be some edge in graph G of $\mathcal{N}$. The next definition describes the key notion of local predicates.

Definition 2.1 *Let* (u, v) *be a link in* $\mathcal{N}$. *We say that a predicate* L *is* local *for* (u, v) *if* L *only involves variables that are part of the state of the* (u, v) *subsystem.*

We shall denote in the sequel a local predicate for (u, v) by L_{uv}. An important property we use frequently is the concept of a closed predicate. Intuitively, a property is closed if it remains true once it becomes true. Formally, a local predicate $L_{u,v}$ is *closed* if whenever $L_{u,v}$ holds in some state of s of the network, then it holds in any possible successor state of s.

We can now explain the concept of local checkability. Suppose we wish a system $\mathcal{N}$ to satisfy some property P, e.g., "each node has a parent pointer so that the set of pointers define a spanning tree of the network." Intuitively, we say that P is *locally checkable* in $\mathcal{N}$ if there exists a collection of local predicates such that P holds if all the predicates in that collection hold. The motivation for introducing this notion, of course, is performance: in a distributed system we can check all link subsystems in parallel in constant time. We formalize the intuitive notion of a local checkability in the following definition.

Definition 2.2 *Let $\mathcal{N}$ be a system, and let P be a property of $\mathcal{N}$. We say that P* is locally checkable *in $\mathcal{N}$ if there exists a collection $\mathcal{L}$ of local predicates, such that*

1. *If at some state s, L_{uv} holds for all $L_{uv} \in \mathcal{L}$, then P holds in s.*
2. *All $L_{u,v} \in \mathcal{L}$ are closed.*

The second item is important for implementation. It stems from the fact that in an asynchronous distributed system it appears to be impossible to check whether an arbitrary local predicate holds *all* the time. What we can do is to "sample" the local subsystem periodically to check if the local property holds. To illustrate this, consider a simple example.

Suppose the network consists of three nodes u, v, and w, and such that v is the neighbor of both u and w. Suppose the property L that we wish to check is the conjunction of two local predicates L_{uv} and L_{vw}. Suppose further that exactly one of the two predicates is always false (and therefore L never holds), but sometimes L_{uv} holds, and in the other times L_{vw} holds. It is possible that whenever we check the (u, v) subsystem we find $L_{u,v}$ true, and whenever we check the (v, w) subsystem we find $L_{v,w}$ true. In this case we may never detect the fact that L does not hold in this execution. We avoid this problem by requiring that $L_{u,v}$ and $L_{v,w}$ be closed.

2.1.1 Implementation of Local Checking

The simplest and most general way to do local checking is to have one of the endpoints (say u) of each (u, v) subsystem periodically take a local *snapshot* [5] of the (u, v) subsystem. To do a snapshot, node u sends a snapshot request to v and stops sending protocol messages to v. When v receives a request, it records its basic state (say s) and sends s in a response to u. When u receives the response, it records its basic state (say r), and the state of the (u, v) subsystem is finally recorded as the combination of s, r, and empty channels. If this link subsystem state does not satisfy the local predicate, a reset request is made.

The snapshot protocol may fail if requests and responses are not properly matched. This can happen if there are spurious packets in the initial state of the system. However, this problem will eventually disappear if we wait T time units between invocations of the snapshot, where T is large enough for all "old" messages to have been delivered.

In many applications, it is sufficient to do checking of local predicates by having each node periodically send its relevant state to all its neighbors. While this is less general than a local snapshot, it works if each local predicate can be separated into two *one-way predicates* for each direction of the link: each one-way predicate refers to variables on the two nodes and only *one* of the two links. We will see an example below in Section 3 (details can be found in [25]).

2.2 Reset: Specification and Implementations

The external interface for a reset protocol at any node u in the network is shown in Figure 1. We have the usual interfaces to send and receive *packets* between neighbors. In addition each node also has interfaces to send and receive *messages* on behalf of external users of the reset. Every message m is drawn from a message alphabet $\Sigma \subset P_{data}$; P_{data} is the data packet alphabet used by the network. Intuitively, messages sent by users to the reset service are relayed between network nodes using packets.

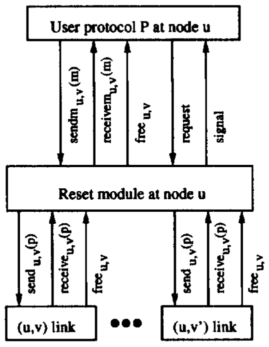

Figure 1: *Interface specification for reset service*

Input action $\text{SENDM}_{u,v}(m)$ allows an external user at u to send a message to neighbor v. Output action $\text{RECEIVEM}_{v,u}(m)$ allows the reset service to deliver a message m from node v to the user at node u. The $\text{FREEM}_{u,v}$ output action indicates that the reset service is ready to accept another message from u to node v. Thus the external interface between a reset service and its users mimics the Data Link interface with packets replaced by messages. The reset service at node u, however, offers two additional actions: an input action REQUEST_u and an output action SIGNAL_u. The REQUEST_u action is used to request a reset, and SIGNAL_u action is used to inform the user at node u that a reset has completed. We call any event of the form SIGNAL_u a *signal event* and any event of the form REQUEST_u a *request event.*

2.3 Specifying Behaviors of a Reset Protocol

We only consider behaviors in which the user of the reset protocol does not send a message unless it first receives a free notification. Formally, we consider only *well formed behaviors*, where a behavior is called well-formed if between any two $\text{SENDM}_{u,v}(*)$ events there is a $\text{FREEM}_{u,v}$ action.

An ideal behavior of a reset protocol should be *timely, consistent, and causal.* Intuitively, a behavior is timely if, in the absence of reset requests, "free" events are delivered in constant time and sent messages are delivered in constant time; it is consistent if there is some symmetrical *mating* relation between signal intervals at neighbors (details provided below); and it is causal if reset signals are only caused by reset requests and reset requests result in reset signals. (Note that causality guarantees *termination*: if reset requests stop, all signal events will stop.)

The most involved part of the reset specification is the specification of consistency. The specification of consistency allows a user to locally reset the user protocol at the instant a signal event is received. Signal events at different nodes are, in general, received at different instants of time. However, consistency ensures that locally resetting the protocol at nodes (when a signal event is received) results in a good global state. To specify consistency, we use the notion of *signal intervals*, defined as follows.

Given an execution of a reset protocol, the signal intervals at a given node are the execution fragments between signal events at that node. Thus the first signal interval at a node u is the time from the start of the execution to the first SIGNAL_u event at the node. Successive signal intervals are demarcated by SIGNAL_u events. If there is a last SIGNAL_u event in an execution, then the interval from the last SIGNAL_u event to the end of the execution is called a *final interval.* Note that each execution can be divided into disjoint signal intervals with respect to each node; of course, the signal intervals induced at different nodes in the same execution will be quite different.

For weak consistency, we require a symmetric relation (called a *mating* relation) between signal intervals at the two nodes such that (i) a signal interval mates with at most one other signal interval, and (ii) the sequence of messages *received* by a node in any interval α is a prefix of the sequence of messages *sent* (by the other node) in the interval β that is the mate of α. If α is a final interval, then β must be final too, and in this case wee require the two sequences to be identical. The last condition is that (iii) for any two distinct intervals at a node that have mates, the mate of the earlier interval precedes (at the other node) the mate of the other interval. A reset that satisfies a weak consistency relation is called a *weak reset.*[1]

For *strong consistency* we also require that the mating relation be transitive. Note that even for weak resets, there is a transitive mating relation between final signal intervals at all nodes. Since the mating relation is symmetric, transitivity allows us to define equivalence classes of signal intervals (with at most one interval per node in each class): this seems to capture the essence of network synchronization.

To specify a reset, we might require that any well-formed behavior be timely, consistent, and causal. Unfortunately, it appears that no reset implementation can stabilize to this set of behaviors! Messages stored in the initial state can result in executions in which all suffixes have some receive event that does not correspond to a send; this violates mating. Thus we say that a protocol $\mathcal{R}$ is a *weak reset* if any well-formed behavior of $\mathcal{R}$ is a suffix of some timely, consistent and weakly causal behavior. Formal details can be found in [25]. We remark that for many applications involving local checking, weak reset protocols suffice.

The *stabilization time* of a reset protocol is the worst-case time before the reset protocol begins to behave correctly even after being started in an arbitrary state. The response time of a reset protocol is the worst-case time it takes for all signal events to disappear after the last request event (assuming there is a last reset request). When we describe a reset protocol as being $O(R)$, we mean that both stabilization and response times are $O(R)$, where a time unit is an upper bound on the time to deliver one message across a link.

Several implementations of stabilizing reset protocols exist. The $O(D)$ protocol of [1] and the $O(n^2)$ protocol of [9] appear to be strong resets. Both require the election of a leader and the time complexities shown exclude the time to compute a leader. The reset protocol of [4] is a $O(n)$ *weak reset.* But it does not require computation of a leader and hence can be used to build an $O(n)$ stabilizing spanning tree protocol (see Section 3). In [3], $O(D)$ reset protocol is

[1] Almost identical forms of synchronization are provided by Data Links [2], virtual circuit protocols ([22]) and transport protocols [23].

described.

3 Spanning Tree Protocol

Spanning tree protocols are widely used, for example in networks containing bridges [18]. The basic idea in virtually all spanning tree algorithms is that nodes report the smallest node ID seen so far (and the shortest distance to this smallest ID node) to their neighbors. Each node then picks as its parent the neighbor that knows of the smallest ID. If more than one neighbor reports the smallest ID, the node picks from among these the neighbor that reports the smallest distance. If two neighbors report the same distance and root ID, then an arbitrary tie-breaker is used to select the parent. A node sets its estimate of the root ID to equal its parent's root ID, and updates its distance from the root to be one plus the parent's distance.

However, in a dynamic network (and in a stabilizing setting), this approach encounters an obstacle known as "ghost roots." This phenomenon occurs whenever the root crashes: its ID, which was the smallest in the system, is still the smallest from the point of view of nodes that do not know of its crash. Even in a static network, the same effect can be caused by initial errors that introduce a root ID lower than the ID of any network node. This ID can potentially remain forever in the system!

The earliest stabilizing spanning tree protocol we know of is due to Perlman [18] and is the basis of the IEEE 802.1 spanning tree protocol. If an old root dies, the protocol must remove information about the old root from the network. In [18] this is done using global timers. The root periodically broadcasts a "hello" message down the tree. If information about a root is not refreshed before a timer expires (whose pre-set time is proportional to the worst case end-to-end network delay) this information is flushed. While the protocol in [18] appears to be self-stabilizing, it requires global time-outs to recover from topological changes, the common failure mode.

The Autonet approach [20] is to use a variant of Finn's unbounded counter protocol [8] to reset the network after each topology change. Together with the spanning tree state, each node also keeps an *epoch counter*. Nodes detecting a topology change reset their state locally and increment the epoch counter. A node receiving a message with a larger epoch counter than its own resets its own state and processes the new message. Thus incrementing the counter forces a new version of the spanning tree protocol. The Autonet approach is faster than that of [18], but it does have a problem when the epoch counter reaches the maximum value because of some catastrophic fault. Thus existing approaches either rely on global timers or are not self-stabilizing.

Our stabilizing spanning tree protocol is extremely similar to the two schemes we have described above. However, it uses a different mechanism (for detecting and recovering from states with ghost roots) that speeds up the stabilization time of the resulting protocol. While we can apply our general method to the AN-1 technique (in which case the reset would reset the epoch counter), there is an even simpler technique that avoids counters altogether and is based on local checking and resets.

The detection mechanism is based on the following observation. Consider an execution of the simple spanning tree protocol that starts with a state in which all nodes are correctly initialized and there are no messages in transit on the links. Now focus on some node. Throughout the execution the node maintains a current estimate of the root ID, and another estimate for its distance from this alleged root. It can be shown that in the course of legal executions, the node's estimate of the root ID never goes up; also while such a root estimate is fixed, the distance estimate never goes up. This property can be cast in the form of a local predicate for each link. If the predicate holds, then the algorithm will produce a spanning tree. This immediately suggests the stabilizing algorithm: whenever the predicate is violated, the node that detects the violation makes a reset request. In the execution that follows the last reset signal, all information will be correct.

3.1 Spanning Tree Protocol Code

We now describe the code for our locally checkable spanning tree protocol. After all local predicates hold, this protocol computes a spanning tree in time proportional to the diameter of the network.

Our locally checkable spanning tree protocol is described in Figure 2. Each node u maintains a parent pointer $parent_u$, current estimate of the root's identity r_u, and a current distance estimate d_u. We denote by (r_u, d_u) the ordered pair at node u. We will use lexicographic ordering for 2-tuples and 3-tuples. For example, $(r_v, d_v) < (r_u, d_u)$ means that either $r_v < r_u$, or that $r_v = r_u$ and $d_v < d_u$. Similarly, $(r_v, d_v, parent_v) < (r_u, d_u, parent_u)$ means that that either $(r_v, d_v) < (r_u, d_u)$, or that $(r_v, d_v) = (r_u, d_u)$ and $parent_v < parent_u$. Each node also maintains a copy (possibly outdated) of the root and distance estimates of each its neighbors. Thus the estimates of neighbor v are stored at u in the variables $r_u[v]$ and $d_u[v]$.

We assume that we are given some bound N on the largest possible distance in the network. For compactness, we let both arrays have an entry for the node itself, in which the default values are "hard-wired": thus $r_u[u] = u$, and $d_u[u] = -1$. Now a node always chooses its own estimate based on the minimum of the estimates of all its neighbors and its own default estimate. Node u chooses its own estimate of the root from this minimum estimate; u also chooses its own estimate of the distance as one plus the distance from the minimum estimate. Thus if u itself is the smallest root, the distance u chooses for itself will be $-1+1=0$, which is as it should be. Thus $d_u[u]$ is set to -1 simply as a sentinel value to avoid an extra case.

Every node u periodically sends its own estimate of root and distance to all its neighbors using an "Announce" packet. The Announce packet is encoded as a tuple $(Announce, r, d)$. When a node u receives an Announce packet from v, u first checks whether the estimate in the packet is greater than the previous estimate stored from v and the distance in the estimate is not already at the maximum possible value. If this is not the case, u stores the received estimate. If the the estimate in the packet is greater than the previous estimate stored from v, then u sets $requestbit_u$ to remember to do a later reset request. If the distance in the estimate is already at the maximum possible value, then u ignores the estimate (accepting it would cause u's distance estimate to overflow). Finally, u resets its sentinel values (in case they were corrupted) and updates its own estimate as the minimum of all neighbor estimates.

A proof of the correctness of a modified version of this protocol can be found in [24]. The main idea is to write a set of local predicates and show that when these local predicates hold, there can be no ghost roots and the protocol computes a spanning tree in time proportional to the network diameter. The key local predicate states that the sequence of estimates sent from any node to a neighbor v are non-increasing. If there are ghost roots, they cannot persist indefinitely without causing some node to receive a greater estimate than its stored value and hence invoke a reset, which initializes the system correctly. For this simple spanning tree protocol, it can be shown than any weak reset is sufficient.

State

$nset_u$	set containing all neighbors of u and u itself
$parent_u$	parent pointer, in $nset_u$
d_u	distance estimate, in range $0..N$, $n < N$
r_u	root estimate
$r_u[v]$	estimate of r_v for v in $nset_u$
$d_u[v]$	estimate of d_v for v in $nset_u$
$free_u[v]$	boolean, true if link to v is free
$requestbit_u$	boolean, remembers pending reset requests

Actions

```
FREEM_{u,v}                                   (*link says its free *)
    Effect: $free_u[v]$ := TRUE

SENDM_{u,v}((Announce, r, d))                 (*send to neighbor $v$*)
    Preconditions:
        $r = r_u$ and $d = d_u$
        $free_u[v]$ = TRUE
    Effect:
        $free_u[v]$ := FALSE

RECEIVEM_{v,u}(Announce, r, d) (*receive estimate from $v$*)
    Effect:
    If $(r,d) \leq (r_u[v], d_u[v])$ then
        If $d < N$                            (*distance at max value ?*)
            $(r_u[v], d_u[v]) := (r, d)$
            $(r_u[u], d_u[u]) := (u, -1)$
    Else $requestbit_u$ := TRUE (*make reset request later*)
    $(r_u, d_u, parent_u)$ :=
                $\min\{(r_u[v], d_u[v]+1, v) : v \in nset_u, d_u[v] < N\}$

REQUEST_u                                     (*request a reset*)
    Preconditions: $requestbit_u$ = TRUE
    Effects: $requestbit_u$ = FALSE

SIGNAL_u                                      (* receive a signal*)
    Effect:
        For all $v \neq u \in nset_u$ do
            $(r_u[v], d_u[v]) := (\infty, \infty)$
            $free_u[v]$ := FALSE
        $(r_u[u], d_u[u]) := (u, -1)$
        $(r_u, d_u, parent_u) := (u, 0, u)$
```

Figure 2: *Spanning tree protocol: code for a node u*

4 Topology Maintenance

In the topology update problem, each network node ascertains the state of its neighboring links and reports this in a so-called Link State packet (LSP). The problem is to broadcast the LSP of each source to all the other nodes in the network. Once this problem is solved, each node has a complete map of the network, and can use this to compute useful information such as shortest path [14, 17] or deadlock free routes [20, 16].

In *intelligent flooding*, used in the new ARPANET protocol [14] and improved by [17, 19], each source independently broadcasts its link state packet using a sequence number, and the highest sequence number is considered to be more recent. We describe the algorithm with respect to a single source s. The overall protocol consists of running several independent versions of the single source protocol, one for each network node.

- Whenever a node v receives an LSP from neighbor u and v has a stored LSP with greater sequence number than u, then v sends its stored LSP back to u. If v's LSP has smaller sequence number, v stores the LSP from u and sends its new LSP to all its neighbors.

- A local checking mechanism verifies periodically that the databases of every pair of neighboring nodes v and u are consistent. If inconsistency is detected, then the more recent information is propagated.

- If the source s receives an LSP with a sequence number N which is higher than the current sequence number at s, then s changes its stored sequence number to $N + 1$, and broadcasts another LSP containing the new sequence number.

With unbounded sequence numbers, even if the system starts with arbitrary sequence numbers, it will stabilize in time proportional to a cross-network delay. This is true since after at most one cross network delay time, the sequence number at the source will be the highest sequence number at the system; and when the source has the highest sequence number, then in one cross-network delay time, all nodes will have the updated data.

However, due to the fact that the sequence number space is finite, this scheme is not self-stabilizing. In [14], for instance, the sequence number space is considered to be cyclic. In a cyclic space it is possible to have three sequence numbers a, b and c such that $a > b > c > a$. Thus, if a transient fault installs three updates with such sequence numbers, the network can loop forever among these updates. In [19], this problem is fixed by using a linear sequence number space, and by waiting for a global timeout when the space is exhausted, in order to allow old updates to die out. The resulting protocol is self-stabilizing, but the recovery time is quite large because of the use of global timers.

Our approach is very simple. Whenever the sequence number reaches its bound at some node, this node makes a reset request. When a reset signal is received at a node other than the source, the node resets its sequence number to 0. When the source receives a signal, the source sets its sequence number to 1 and broadcasts its most recent LSP.

The simplest approach is to use a single reset that resets the information about all sources. If the counter is sufficiently large, the reset should happen sufficiently rarely so that the global disruption caused by such a reset is not a factor. It is also easy to have separate reset procedures for each source, but that requires separate state for each such reset protocol.

The solution in [17] and [19] has one additional mechanism. Periodically, at intervals of some global timer (say every 10 minutes),e ven if the LSP information has not changed, the source increments the sequence number by one to produce a new LSP, and then broadcasts it to its neighbors. The purpose of this rule is twofold. First, it provides another level of defense against catastrophic faults as the source is periodically rebroadcasting its LSP information. More importantly, it is used to get rid of obsolete information from nodes that have died. After waiting for say half an hour without getting a LSP from a source s, a node can discard the LSP from s as being useless. Thus this mechanism is used for garbage collection.

We prefer to retain the garbage collection scheme because of its simplicity. However, with our modifications, ***the timer that controls periodic retransmission of LSPs does not control the self-stabilization recovery times but only affects the time to do garbage collection.*** Thus this timer can be set to be a high enough value to cover the worst case end-to-end delay in the largest possible network (say 1 hour). If local checking is done in the order of seconds, and the actual delay through the network is also in the order of seconds, then the recovery time from arbitrary faults can be in the order of seconds. However, if a source s is removed from the network, it will take 1 hour before other nodes can reclaim the storage taken up by s's LSP.

5 Conclusion and Discussion

Philosophically, our aim in this paper is to demonstrate that simplicity and robustness need not be sacrificed in practical protocols. We argue that elegant unbounded-registers protocols can be used in a realistic setting, without making them vulnerable to transient faults. Many popular systems use either specially tailored bounded-register protocols or rely on inefficient global timers. We described a method by which one can transform an unbounded-register protocol (typically very simple) into a robust bounded-register protocol with practically the same behavior. Another example of the methodology of this paper is the design of a self-stabilizing network synchronization protocol [3] which can be used to convert synchronous protocols into stabilizing asynchronous equivalents. We first designed a stabilizing protocol that worked with unbounded counters, and then transformed it using a reset protocol.

References

[1] A. Arora and M. 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.

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

[3] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *Proc. 25th ACM Symp. on Theory of Computing*, Oct. 1993.

[4] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, Oct. 1991.

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

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

[7] D. Dolev and N. Shavit. Bounded concurrent time stamps systems are constructible. In *Proc. 21st ACM Symp. on Theory of Computing*. ACM SIGACT, ACM, 1989.

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

[9] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, Aug. 1990.

[10] L. Kleinrock and F. Kamoun. Hierarchical routing for large networks; performance evaluation and optimization. *Computer Networks*, 1:155–174, 1977.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[12] L. Lamport and N. Lynch. Distributed computing. Chapter of Handbook on Theoretical Computer Science. Also, to be published as Technical Memo MIT/LCS/TM-384, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridg e, MA, 1989.

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

[14] J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Communications*, COM-28(5):711–719, May 1980.

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

[16] 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.

[17] R. Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, Dec. 1983.

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

[19] R. Perlman, G. Varghese, and A. Lauck. Reliable broadcast of information in a wide area network. *US Patent 5,085,428*, Feb. 1992.

[20] T. Rodeheffer and M. Schroeder. Automatic reconfiguration in the Autonet. Proceedings of the 14th Symposium on Operating Systems Principles, Nov 1993.

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

[22] J. M. Spinelli. Reliable communication. Ph.d. thesis, MIT, Lab. for Information and Decision Systems, Dec. 1988.

[23] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2d.edition edition, 1989.

[24] G. Varghese. Self-stabilization by counter flushing. To appear as a Washington University Technical Report, 1993.

[25] G. Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.