Chicago Journal of Theoretical Computer Science

The MIT Press

Volume 1997, Article 4
19 December 1997

ISSN 1073–0486. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493 USA; (617)253-2889; *journals-orders@mit.edu, journals-info@mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://mitpress.mit.edu/CJTCS/*
- *http://www.cs.uchicago.edu/publications/cjtcs/*
- *ftp://mitpress.mit.edu/pub/CJTCS*
- *ftp://cs.uchicago.edu/pub/publications/cjtcs*

The *Chicago Journal of Theoretical Computer Science* is abstracted or indexed in *Research Alert,*® *SciSearch,*® *Current Contents*®*/Engineering Computing & Technology*, and *CompuMath Citation Index.*®

The *Chicago Journal of Theoretical Computer Science* is a peer-reviewed scholarly journal in theoretical computer science. The journal is committed to providing a forum for significant results on theoretical aspects of all topics in computer science.

Superstabilizing Protocols for Dynamic Distributed Systems[*]

Shlomi Dolev Ted Herman

19 December, 1997

Abstract

Abstract-1 Two aspects of distributed-protocol reliability are the ability to recover from transient faults and the ability to function in a dynamic environment. Approaches for both of these aspects have been separately developed, but have revealed drawbacks when applied to environments with both transient faults and dynamic changes. This paper introduces definitions and methods for addressing both concerns in system design.

Abstract-2 A protocol is *superstabilizing* if it is (1) self-stabilizing, meaning that it is guaranteed to respond to an arbitrary transient fault by eventually satisfying and maintaining a *legitimacy* predicate, and it is (2) guaranteed to satisfy a *passage* predicate at all times when the system undergoes topology changes starting from a legitimate state. The passage predicate is typically a safety property that should hold while the protocol makes progress toward reestablishing legitimacy following a topology change.

Abstract-3 Specific contributions of the paper include: the definition of the class of superstabilizing protocols; metrics for evaluating superstabilizing protocols; superstabilizing protocols for coloring and spanning tree construction; a general method for converting self-stabilizing protocols into superstabilizing ones; and a generalized form of a self-stabilizing topology update protocol which may have useful applications for other research.

[*] A preliminary version of this work was presented at the 2nd Workshop on Self-Stabilizing Systems, Las Vegas, Nevada, May 1995.

1 Introduction

1-1 The most general technique enabling a system to tolerate arbitrary transient faults is *self-stabilization*: a protocol is self-stabilizing if, in response to any transient fault, it converges to a legitimate state in finite time. The characterization of legitimate states, given by a legitimacy predicate, specifies the protocol's function. Such protocols are generally evaluated by studying the efficiency of convergence, which entails bounding the time of convergence to a legitimate state following a transient fault. Other aspects of convergence, for instance, safety properties, are of less interest, because arbitrary transient faults can falsify any nontrivial safety property.

1-2 The model of a *dynamic* system is one where communication links and processors may fail and recover during normal operation. Protocols for dynamic systems are designed to contend with such failures and recoveries without global reinitialization. These protocols consider only global states that are reachable from a predefined initial state under a *constrained sequence of failures*; under such an assumption, the protocols attempt to handle failures with as few adjustments as possible. Thus, whereas self-stabilization research largely ignores the behavior of protocols between the time of a transient fault and restoration to a legitimate state, dynamic protocols make guarantees about behavior at all times (during the period between a failure event and the completion of necessary adjustments).

1.1 Superstabilization

1.1-1 Superstabilizing protocols combine benefits of both self-stabilizing and dynamic protocols. We retain the idea of a legitimate state, but partition illegitimate states into two classes, depending on whether or not they satisfy a *passage* predicate. Roughly speaking, a protocol is superstabilizing if: (1) it is self-stabilizing, and (2) when it is started in a legitimate state and a topology change occurs, the passage predicate holds and continues to hold until the protocol reaches a legitimate state.

1.1-2 Superstabilization is specified with respect to the class of topology changes for which a desired passage predicate can be maintained during convergence to legitimacy. Since a legitimacy predicate is dependent on system topology, a topology change will typically falsify legitimacy. The passage predicate must therefore be weaker than legitimacy, but strong enough to be useful; ideally, the passage predicate should be the strongest predicate that holds when a legitimate state undergoes any of a class of topology change events. One example for a passage predicate is the existence of at most one token in a mutual exclusion task: whereas in a legitimate state exactly one token exists, a processor crash could eliminate the token yet not falsify the passage predicate. Similarly, for the leader election task, the passage predicate could specify that at most one leader exists. A useful passage predicate can thus represent a critical safety aspect of the task in question. A superstabilizing protocol is more stable[1] than a protocol that is only self-stabilizing: at a legitimate state, the system maintains stability with respect to the passage predicate, even when perturbed by a topology change.

1.1-3 Superstabilizing protocols are evaluated in several ways. Of interest are the worst-case convergence time, i.e., the time required to establish a legitimate state following either a transient fault or a topology change, and the scope of the convergence in terms of how much of the network's data must be changed as a result of convergence. We classify superstabilizing protocols by the following complexity measures:

- stabilization time—the maximum amount of time it takes for the protocol to reach a legitimate state;
- superstabilization time—the maximum amount of time it takes for a protocol starting from a legitimate state, followed by a single topology change, to reach a legitimate state; and
- adjustment measure—the maximum number of processors that must change their local states, upon a topology change from a legitimate state, so that the protocol is in a legitimate state.

1.2 Background and Motivation

1.2-1 Many distributed protocols have been designed to cope with continuous dynamic changes [AAG87, AGH90, AM92, AGR92]. These protocols make certain assumptions about the behavior of processors and links during failure and recovery; for instance, most of them do not consider the possibility of processor crashes, and they assume that every corrupted message is identified and discarded. If failures are frequent, these restrictive assumptions can be too optimistic.

[1] This stronger property of stability motivates the term *superstabilizing*.

1.2-2 A number of researchers [DIM93, KP93, APSV91, AG94b] suggest a self-stabilizing approach for dealing with dynamic systems. In a self-stabilizing approach, a state following a topology change is seen as an inconsistent state from which the system will converge to a state consistent with the new topology. Although self-stabilization can handle dynamic systems, the primary goal of a self-stabilizing protocol is to recover from transient faults. This view has influenced the design and analysis of self-stabilizing protocols; for instance, for a correct self-stabilizing protocol, there are no restrictions on the behavior of the system *during* the convergence period—the only guarantee is convergence to a legitimate state.

1.2-3 When considering the treatment of a dynamic system, self-stabilization differs from the dynamic protocols cited above in the way that topology changes are modeled. The dynamic protocols assume that topology changes are *events* signaling changes on incident processors. Self-stabilizing protocols take a necessarily more conservative approach that is entirely state-based: a topology change results in a new state, from which convergence to a legitimacy is guaranteed, with no dependence on a signal. Yet when the system is in a legitimate state and a fault happens to be a detected event, can the behavior during the convergence be constrained to satisfy some desired safety property? For instance, is it possible in these situations for the protocol to maintain a "nearly legitimate" state during convergence?

1.2-4 The issue can be motivated by considering the problem of maintaining a spanning tree in a network. Suppose the spanning tree is used for virtual circuits between processors in the network. When a tree link fails, the spanning tree becomes disconnected; yet virtual circuits entirely within a connected component can continue to operate. We would like to restore the system to have a spanning tree so that existing virtual circuits in the connected components remain operating; thus a least-impact legitimate state would be realized by simply choosing a link to connect the components.

1.2-5 One thesis of this paper is that self-stabilizing protocols can be designed with dynamic change in mind to improve response. Self-stabilizing protocols proposed for dynamic systems do not use the fact that a processor can detect that it is recovering following a crash; consequently, there is no possibility of executing an "initialization" procedure during this recovery.[2] A key obser-

[2] Moreover, some systems *do* provide a "start" signal. Self-stabilizing protocols do not specify any special starting state or action upon receiving such a signal. Note that a computation invoked by a start signal resembles the case of a dynamic system starting with all processors simultaneously recovering from a crash.

vation for this paper is that a topology change is *usually* a detectable event, and in cases where a topology change is not detected, we use self-stabilization as a fallback mechanism to deal with the change. The remainder of the paper illustrates aspects of superstabilization with selected protocols, and a general method that converts self-stabilizing protocols into superstabilizing protocols.

2 Dynamic Systems

2-1 A system is represented by a graph where processors are nodes and links are (undirected) edges. An edge between two processors exists if and only if the two processors are *neighbors*; processors may only communicate if they are neighbors. Each processor has a unique identifier taken from a totally ordered domain. We use p, q, and r to denote processor identifiers. Processors communicate using registers, however, application of the model to a message-passing system is intended; we outline an implementation of the register model in terms of message-based constructions in [DIM97, DH95].

2-2 Associated with each processor p are code, local variables, a program counter, a shared register, and an input variable N_p, which is a list of processors q that are neighbors of p. Invariantly, neighborhoods satisfy $p \notin N_p$ and $q \in N_p \Leftrightarrow p \in N_q$. A processor can read from and write to its own shared register, but may only read shared registers belonging to neighboring processors. The code of a processor is a sequential program; a program counter is associated with each processor. An *atomic step* of a processor (in the sequel referred to as steps) consists of the execution of one statement in a program. In one atomic step, a processor performs some internal computation and, at most, one register operation. A processor has two possible register operations, read and write.

2-3 Our model specifies that a step consists of a statement execution; we have in mind a conventional instruction-execution architecture, where each statement corresponds to some low-level code. However, to make the presentation of protocols concise, we give descriptions at a higher level in terms of programs with assignment statements and control structures (**forall**, **do**, etc.); it should be understood that these descriptions can be resolved into lower-level programs where statements translate into atomic steps. In the protocol presentations, we also use the convention that advancing the program counter beyond the last statement of a program returns the program counter to the program's first statement; thus each program takes the form of an infinite loop.

2-4 Local processor variables are of two types: variables used for computations, and *field image* variables. The former are denoted using unsubscripted variable names such as x, y, and A. The field image variables refer to fields of registers; these variables are subscripted to refer to the register location. For instance, e_p refers to a field of processor p's register, and y_q refers to a field of processor q's register. Program statements that assign to field images or use field images in calculations are not register operations: the field image is essentially a cache of an actual register field. A processor p's $\mathsf{read}(q)$ operation, defined for $q \in N_p$, atomically reads the register of processor q and assigns all corresponding field images (e.g., e_q, y_q, etc.) at processor p. A write operation atomically sets all fields of p's register to current image values. For convenience, we also permit a local calculation to specify field image(s) with a write statement; for instance, $\mathsf{write}(e_p := 1)$ assigns to field image e_p and then writes to p's register.

2-5 The state of a processor p fully describes the values of its local variables, program counter, shared register, and its neighborhood N_p (although a processor cannot change its neighborhood, it is convenient to include N_p in the state of p for subsequent definitions). In the sequel, we occasionally refer to the state of a processor as a *local state*. The state of the system is a vector of states of all processors; a system state is called a *global state*. For a global state σ and a processor q, let $\sigma[q]$ denote the local state of q in state σ. A *computation* is a sequence of global states $\Theta = (\theta_1, \theta_2, \ldots)$ such that for $i = 1, 2, \ldots$ the global state θ_{i+1} is reached from θ_i by a single step of some processor. A *fair* computation is a computation that is either finite or contains infinitely many steps of each (noncrashed) processor.

2-6 We write $\sigma \vdash \mathcal{Q}$ to denote that global state σ satisfies a predicate $\mathcal{Q}$. Suppose $\mathcal{P}$ is a predicate that encodes some property of interest for our system; for instance, $\mathcal{P}$ can specify that exactly one processor has a token for mutual exclusion. A legitimacy predicate $\mathcal{L}$ is typically specified with respect to $\mathcal{P}$: whereas $\mathcal{P}$ is concerned with those aspects of a state to determine whether or not the property of interest holds, the predicate $\mathcal{L}$ specifies permissible values of all register fields, program counters, and local variables so that $\mathcal{P}$ remains invariantly true in a computation.

2-7 A system *topology* is a specific system configuration of links and processors. Each processor can determine the current status of its neighborhood from its local state (via N_p); thus the system topology can be extracted by a program from a global state of the system. Let $\mathcal{T}.\alpha$ denote the topology for a given global state α. Dynamic changes transform the system from one topology $\mathcal{T}.\alpha$ to another topology $\mathcal{T}.\beta$ by changing neighborhoods and possibly removing or adding processors.

2-8 A topology change *event* is the removal or addition of a single neighborhood component (link, processor, or processor and subset of its incident links), together with the execution of certain atomic steps specified in the sequel. Topology changes involving numerous links and processors can be modeled by a sequence of single change events. The crash of processor p is denoted crash_p; the recovery of processor p is denoted recov_p; crash_{pq} and recov_{pq} denote link failure and recovery events, respectively. In our model, a processor crash and a link crash are indistinguishable to a neighbor of the event: if p and q are neighbors and crash_p occurs, then we model this event by crash_{pq} with respect to reasoning about processor q. Similarly, a recov_p event is indistinguishable from a recov_{pq} event with respect to reasoning about a neighbor q of p. We say that a topology change event $\mathcal{E}$ is *incident* on p if $\mathcal{E}$ is recov_p, crash_{pq}, or recov_{pq}. We extend this definition to be symmetric: $\mathcal{E}$ is incident on p if and only if p is incident on $\mathcal{E}$.

2-9 For most of the protocols presented in this paper, each processor is equipped with an *interrupt statement,* which is a statement concerned with adjusting to topology change. A topology change $\mathcal{E}$ incident on p causes the following to atomically occur at p: the input variable N_p is changed to reflect $\mathcal{E}$, the interrupt statement of the protocol is atomically executed, and p's program counter is set to the first statement of its program. Note that if $\mathcal{E}$ is incident on numerous processors, then all incident neighborhoods change to reflect $\mathcal{E}$, and all processors execute the interrupt statement atomically with event $\mathcal{E}$. Thus the transition by $\mathcal{E}$ from $\mathcal{T}.\alpha$ to $\mathcal{T}.\beta$ changes more than the neighborhoods; states α and β also differ in the local states of processors incident on $\mathcal{E}$, due to execution of interrupt statements at these processors.

2-10 A *trajectory* is a sequence of global states in which each segment is either a fair computation or a sequence of topology change events. For purposes of reasoning about self-stabilization, we follow the standard method of proving properties of computations, not trajectories. Dynamic change is handled indirectly in this approach: following an event $\mathcal{E}$, if there are no further changes for a sufficiently long period, the protocol self-stabilizes in the computation following $\mathcal{E}$ in the trajectory.

3 The Superstabilization Paradigm

3-1 The definition of superstabilization takes the idea of a "typical" change into account by specifying a class Λ of topology change events. A self-stabilizing protocol is superstabilizing with respect to events of type Λ, if starting from a legitimate state followed by a Λ-event, the passage predicate holds continuously:

Definition 1 *A protocol P is* superstabilizing *with respect to Λ if and only if P is self-stabilizing, and for every trajectory Φ beginning at a legitimate state and containing a single topology change event of type Λ, the passage predicate holds for every $\sigma \in \Phi$.*

3-2 Although Definition 1 considers trajectories with a single change, we emphasize that the intent is to handle trajectories with multiple changes (each change is completely accommodated before the next change occurs). Our definition could be modified to state this explicitly, however, we have chosen this simpler form to streamline presentations.

3-3 A primary motivation for superstabilization is the notion of a "low-impact" reaction by a protocol to dynamic change. Intuitively, this means that changes necessary in response to dynamic change should affect relatively few processors and links. To formalize this idea, we introduce an adjustment measure. To define an adjustment, we return to the concept of legitimacy and a predicate $\mathcal{P}$ that effectively characterizes the legitimacy predicate $\mathcal{L}$. Let $var(\mathcal{P})$ be the minimal collection of variables and fields upon which $\mathcal{P}$ depends. Call $\mathcal{O}$ the state-space ranging only over the $var(\mathcal{P})$ data. The expression $\delta[\mathcal{O}]$ denotes a system state projected onto the $\mathcal{O}$ state-space. Next we consider a function $\mathcal{F} : \mathcal{O} \longrightarrow \mathcal{O}$. Function $\mathcal{F}$ maps states $\delta[\mathcal{O}]$ to states $\sigma[\mathcal{O}]$ that satisfy $\sigma \vdash \mathcal{L}$, where δ is any state that can be obtained from a legitimate state followed by a Λ-topology change $\mathcal{E}$. The idea is that $\mathcal{F}$ represents the strategy of a superstabilizing protocol in reacting to an event $\mathcal{E}$, namely, choosing a new legitimate state following dynamic change. We rank a function $\mathcal{F}$ by means of an adjustment measure $\mathcal{R}$. The adjustment measure $\mathcal{R}$ is the maximum number of processors having different $\mathcal{O}$-states between $\rho[\mathcal{O}]$ and $\mathcal{F}(\rho[\mathcal{O}])$, taken over all states ρ derived from some state $\delta \vdash \mathcal{L}$ followed by some change event $\mathcal{E} \in \Lambda$. A definition of $\mathcal{F}$ with a small adjustment measure $\mathcal{R}$ implies that few adjustments are necessary in response to a topology change.

3-4 A superstabilizing protocol *respects* $\mathcal{F}$ if the protocol implements $\mathcal{F}$, meaning that it responds to a dynamic change by some computation Θ taking the system from a state $\rho[\mathcal{O}]$ to a state $\mathcal{F}(\rho[\mathcal{O}])$, and changes $\mathcal{O}$-states at the minimum number of processors necessary in order to establish the new legitimate state given by $\mathcal{F}$. If a protocol respects $\mathcal{F}$, then we say that the protocol has adjustment measure $\mathcal{R}$.

3-5 The notion of adjustment measure can be regarded informally as a distance, in terms of processor states, between legitimate states of different configurations. It is plausible to consider other definitions of distance upon which optimal adjustment could be based. For example, if the state change of a particular processor could cause more damage than changing the states of all other processors, a weighted definition of adjustment measure would be appropriate. The general technique described in Section 6 can be easily modified to support alternative definitions of adjustment measure.

3-6 To describe the time complexity of a protocol, the idea of a *cycle* is introduced. A cycle for a processor p is a minimal sequence of steps in a computation so that an iteration of the protocol at processor p executes the program for p from the first to the last statement. All the programs of this paper are constructed so that a processor p's cycle consists of reading all of p's neighbor registers, performing some local computation, and writing into p's register. The time-complexity of a computation is measured by *rounds*, defined inductively as follows. Given a computation Φ, the first round of Φ terminates at the first state at which every processor has completed at least one cycle; round $i+1$ terminates after each processor has executed at least one cycle following the termination of round i.

3-7 The stabilization time of a protocol is the maximum number of rounds it takes for the protocol to reach a legitimate state, starting from an arbitrary state. The superstabilization time is the maximum number of rounds it takes for a protocol starting from an arbitrary legitimate state σ, followed by an arbitrary Λ-change event $\mathcal{E}$, to again reach a legitimate state.

4 Superstabilizing Coloring

4-1 This section exercises the definitions and notation developed in Sections 2 and 3 for a simple allocation problem. Let $\mathcal{C}$ be a totally ordered set of colors satisfying $|\mathcal{C}| \geq 1 + \Delta$, where Δ is an upper bound on the number of neighbors a processor has in any trajectory. Each processor p has a register field $color_p$.

```
Self-Stabilizing Section:
S1  $A,\ B\ :=\ \emptyset,\ \emptyset$
S2  forall $q \in N_p$
S3      do
            read($q$);
            $A\ :=\ A \cup color_q$;
            if $q > p$ then $B\ :=\ B \cup color_q$
        od
S4  if $(\ (color_p = \perp\ \wedge\ \perp \notin B)\ \vee\ (color_p \neq \perp\ \wedge\ color_p \in B)\ )$
    then $color_p\ :=\ choose(\mathcal{C} \setminus A)$
S5  write
Interrupt Section:
E1  write( if ( $\mathcal{E} = \mathsf{recov}_p\ \vee\ (\mathcal{E} = \mathsf{recov}_{pq}\ \wedge\ p > q)$ )
            then $color_p\ := \perp$ )
```

Figure 1: Superstabilizing coloring protocol for processor p

The predicate $\mathcal{P}$ of interest is: $color_p \in \mathcal{C}$ for every processor p, and no two neighboring processors have equal colors. A legitimate state for the coloring protocol is any state such that (1) predicate $\mathcal{P}$ is satisfied, and (2) for each computation that starts in such a state, no processor changes color in the computation. To define the passage predicate, we extend the domain of a color field to include $\perp \notin \mathcal{C}$. The passage predicate is: $color_p \in \mathcal{C} \cup \{\perp\}$ for every processor p and, for any neighboring processors p and q, $color_p = color_q$ if and only if $color_p = \perp$.

4-2 Figure 1 presents a protocol for the coloring problem. The function $choose(S)$ selects the minimum color from a set S (and is undefined if S is empty). The protocol of Figure 1 has two parts: one part is a self-stabilizing protocol, modified to deal with the $\perp$ element; the other part lists the interrupt that handles topology change events. Note that although $\perp$ is introduced to contend with topology changes, one must then consider the possibility of an initial state (for example, due to a transient fault) in which $color_p = \perp$ holds for every processor p. The self-stabilizing section perpetually scans for a color conflict with the set of neighboring processors having a larger identifier. The interrupt statement writes to the register, conditionally changing the $color_p$ field in case the topology change event is a restart of the processor or a link.

Theorem 1 *The coloring protocol is self-stabilizing and converges in $O(n)$ rounds.*

Proof of Theorem 1 We show by induction on the number of processors, that following round i, $0 \leq i \leq n$, the i largest-identifier processors have permanent, non-$\perp$ color assignments such that no conflict with a neighbor of higher identity exists among these i processors. The basis for the induction is trivial, since the empty set of processors satisfies the assertion. Next, suppose the claim holds following round k. We examine the effect of round $(k+1)$ with respect to processor r, where r is the $(k+1)^{\text{th}}$ largest processor identifier. In this round, processor r chooses some non-$\perp$ color differing from any color of a neighboring processor with larger identity. The choice is deterministic, based on the colors of the larger identities. By hypothesis, these larger identity color assignments are permanent, so following round $(k+1)$ and for all subsequent rounds, processor r's color is fixed and differs from the colors of all neighbors of larger identity. Thus after $(n+1)$ rounds, all processors have permanent color assignments.

Proof of Theorem 1 □

The class of topology events considered for the protocol is $\Lambda(k)$, $0 \leq k \leq \Delta$, which includes any crash event, any link recov_{pq} event, and any recov_p event, subject to the restriction that at most k links incident on p recover at the same instant that p recovers; thus, $\Lambda(0)$ allows only processor or link recovery events that are not simultaneous, whereas $\Lambda(\Delta)$ includes the possibility of a processor and all its links recovering as a single event.

4-3

Theorem 2 *The coloring protocol is superstabilizing with a superstabilizing time of $O(k)$ and an adjustment measure $\mathcal{R} = (k+1)$, where $\Lambda(k)$ is the class of topology change events.*

Proof of Theorem 2 In the case of any crash event, the protocol remains in a legitimate state. In the case of a recov event, for any new link introduced by the event, one or both of the incident processors has color $\perp$ as a result; thus the passage predicate holds. Moreover, at most $(k+1)$ processors can have color $\perp$ as a result of the recov event; by an argument similar to that given in the proof of Theorem 1, a legitimate state is obtained in $O(k)$ rounds, and the passage predicate holds invariantly in the computation; the adjustment measure $\mathcal{R} = (k+1)$ follows from the fact that only those processors incident on the change event adjust color during the period of superstabilization.

Proof of Theorem 2 □

4-4 The coloring protocol illustrates qualitative and quantitative aspects of superstabilization. The qualitative aspect is illustrated by the fact that the convergence following a topology change does not violate a passage predicate. This ensures better service to the user when no catastrophe takes place (i.e., in the absence of a severe transient fault). Quantitative aspects can be seen by the $O(k)$ convergence time and adjustment measure. The same protocol, when started in an arbitrary initial state induced by a transient fault, might take $O(n)$ rounds to converge, and a processor could change colors $O(n)$ times during this convergence. Indeed, if the superstabilizing component (namely, the interrupt statement) of the protocol is removed, then $O(n)$ rounds can be required for convergence following even a single topology change event starting from a legitimate state.

5 The Superstabilizing Tree

5-1 Constructing a spanning tree in a network is a basic task for many protocols. Several distributed reset procedures, including self-stabilizing ones, rely on the construction of a rooted spanning tree to control synchronization. All existing deterministic self-stabilizing algorithms for constructing spanning trees rely on processor or link identifiers to select, for example, a shortest-path tree or a breadth-first search tree. In a dynamic network, a change event can invalidate an existing spanning tree and require that a new tree be computed. Although computation is required when a change event crash_{pq} removes one of the links in the current spanning tree, one would hope that a change event $\mathsf{recover}_{pq}$ would require no adjustment to an existing spanning tree. Most of the self-stabilizing spanning tree algorithms we know (e.g., [DIM93, AG94b, AK93]) construct a BFS (i.e., breadth-first search) or DFS (i.e., depth-first search) tree and thus require, in some cases, recomputation of the tree when a link recovers, regardless of whether or not the network currently has a spanning tree. The reason is that a processor cannot locally

```
Self-Stabilizing Section:
S1   $x, y := \infty, \bot$
S2   forall $q \in N_p$
S3       do
             read($q$);
             if $x > (d_q + w_{pq})$ then $x, y := (d_q + w_{pq}), q$
         od
S4   if $p = r$ then $d_p, t_p := 0, r$ else $d_p, t_p := x, y$
S5   write
Interrupt Section:   skip
```

$$\text{where } w_{pq} = \begin{cases} 1 & \text{if } t_p = q \\ n & \text{if } t_p \neq q \end{cases}$$

Figure 2: Superstabilizing tree protocol for processor p

"know" that the system has stabilized, and must make a deterministic choice of edges to be included in the tree. We propose a superstabilizing approach to tree construction, which is a variant of the algorithm proposed in [CYH91]. The protocol given in this section successfully "ignores" all dynamic changes that add links to an existing spanning tree or crash links not contained in the tree.

5-2 All trajectories considered in this section are free of crash_p or $\mathsf{recover}_p$ events; the number of processors remains fixed at n, and we give every processor access to the constant n. We also suppose that the network remains, at all states in a trajectory, connected.

5-3 The basic idea of the protocol is the construction of a least-cost path tree to a processor r, which is designated as the root of the tree. The key point of the protocol lies in the definition of link costs. Each link is assigned a cost in such a way that links that are part of the tree have low cost, whereas links outside the tree have high cost. Each processor p has two register fields, t_p and d_p. The field t_p ranges over identifiers of processors, and represents the parent of p in the tree (by convention, we let $t_r = r$). The register d_p contains a non-negative integer representing the cost of a path from p to the root r.

5-4 Figure 2 shows the code of the superstabilizing spanning tree protocol. The predicate $\mathcal{P}$ of interest for the tree protocol is that $(\forall p, q :\; p \neq r \;\wedge$

$t_p = q : q \in N_p$), and that the collection of t_p variables $\{t_p \mid p \neq r\}$ represents a spanning, directed tree rooted at r. A legitimate state for the tree protocol is any state such that (1) predicate $\mathcal{P}$ is satisfied, and (2) for each computation that starts in such a state, no processor changes a t_p variable in the computation.

Theorem 3 *The spanning tree protocol self-stabilizes in $O(n)$ rounds.*

(See Appendix A for the proof.)

5-5 We define the class of change events Λ for purposes of superstabilization to be any recov_{pq} event or any crash_{pq} event such that neither $t_p = q$ nor $t_q = p$ holds at the moment of the crash_{pq} event; i.e., the p-q link is not a link in the current tree. The passage predicate for the superstabilization property is identical to $\mathcal{P}$.

Theorem 4 *The spanning tree protocol is superstabilizing for the class Λ, with superstabilization time $O(1)$ and adjustment measure $\mathcal{R} = 1$.*

(See Appendix A for the proof.)

6 General Superstabilization

6-1 This section introduces a general method for achieving superstabilization with respect to the class Λ of single topology changes. Our general method can be seen as a compiler that takes self-stabilizing protocol P and outputs a new protocol P^s that is both self-stabilizing and superstabilizing. This is done by modifying protocol P and superimposing a new component, called the *superstabilizer*. The superstabilizer uses, as a tool, a self-stabilizing update protocol. The following section describes our update protocol, after which we give an overview of the superstabilizer.

6.1 The Update Protocol

6.1-1 To simplify the presentation of our general method for superstabilizing protocols, we employ a self-stabilizing update protocol. We view the update protocol as the simplest and clearest self-stabilizing protocol for a large class of tasks including leader election, topology update, and diameter estimation. Our update protocol enjoys a number of properties not directly required for general superstabilization. Some of these properties are stated and proved in Appendix B, since they may have useful application to other results on stabilization. We remark that our update protocol works in unidirectional systems in which the existence of a communication link from one node to another does not imply the existence of a link in the opposite direction (e.g., [AG94a, AB97]).

6.1-2 To describe the task of the update protocol, suppose every processor p has some field image x_p; for the moment, we consider x_p to be a constant. The *update* problem is to broadcast each x_p to all processors. This problem is called the *topology update* when the field x_p contains all the local information about p's links and network characteristics. Many dynamic systems are already equipped with a topology update protocol that notifies processors of the current topology; in such instances, our general method acts as an extension to this existing topology update. An optimal time ($\Theta(d)$ round) self-stabilizing solution to the topology update is given in [SG89, Dol93]. To ensure a desired deterministic property of the protocol, we assume that the neighborhood of a processor N_p is represented as an ordered list.

6.1-3 Let each processor p have, in addition to x_p, a field e_p, where e_p contains three tuples of the form $\langle q, u, k \rangle$, in which q is a processor identifier, u is of the same type as x_p, and k is a non-negative integer. Let $\mathit{dist}_{\mathcal{T}}(p, q)$ be the minimum number of links contained in a path between processors p and q in topology $\mathcal{T}$ (in arguments where the topology is understood from the context, we write $\mathit{dist}(p, q)$); the third component of the tuple is intended to represent the *dist* value between p and the processor named in the tuple's first component. We make some notational conventions in dealing with tuples: with respect to a given (global) state, $\langle q, x_q, k \rangle$ is a tuple whose second component contains the current value of field x_q. In proofs and assertions, we specify tuples partially: $\langle q, , \rangle \in e_p$ is the assertion that processor p's e field contains a tuple with q as its first component. Each processor uses local variables A and B that range over the same set of tuples that e_p does. For field image e_p and set variables A and B, we assume that set operations are implemented so that computations on these objects are deterministic.

6.1-4 The update protocol's code uses the following definitions. Let $\mathsf{processors}(A)$ be the list of processor identifiers obtained from the first components of the tuples in A. Let $\mathsf{mindist}(q, A)$ be the first tuple in A having a minimal third component of any tuple whose first component is q (in case no matching tuple exists, then $\mathsf{mindist}$ is undefined). Define $A \mathbin{\backslash\backslash} \langle q, *, * \rangle$ to be the list of tuples obtained from A by removing every tuple whose first component is q.

```
C1   $A, B := \emptyset, \emptyset$
C2   forall $q \in N_p$ do read($q$);  $A := A \cup e_q$ od
C3   $A := A \setminus\setminus \langle p, *, * \rangle$;  $A := A{+}{+}\langle *, *, 1 \rangle$
C4   forall $q \in$ processors($A$) do $B := B \cup \{$mindist($q, A$)$\}$ od
C5   $B := B \cup \{\langle p, x_p, 0 \rangle\}$;  $e_p :=$ initseq($B$)
C6   write
```

Figure 3: Update protocol for processor p

Define $A{+}{+}\langle *, *, 1 \rangle$ to be the list of tuples obtained from A by incrementing the third component of every tuple in A. Define initseq(A) by the following procedure: first sort the tuples of A in ascending order of the third element of a tuple; then, from this ordered sequence of tuples, compute the maximum initial prefix of tuples with the property: if $\langle q, u, k \rangle$ and $\langle q', u', k' \rangle$ are successive tuples in the prefix, then $k' \leq k + 1$. Then initseq(A) is the set of tuples in this initial prefix.

6.1-5 For the update protocol, we define a *distance-stable* state to be any state for which: (1) each processor p has exactly one tuple $\langle q, y, dist(p, q) \rangle$ in its e_p field for every processor q in the network reachable by some path from p in the current topology; (2) e_p contains no other tuples; and (3) each computation that starts in such a state preserves (1) and (2). A *legitimate* state for the update protocol is a distance-stable state in which requirement (1) is strengthened to: each processor p has exactly one tuple $\langle q, x_q, dist(p, q) \rangle$ in its e_p field for every processor q—in other words, the x field images are accurate. Figure 3 presents the protocol.

Theorem 5 *The update protocol of Figure 3 self-stabilizes in $O(d)$ rounds.*

(See Appendix B for proof.)

6.1-6 A corollary of self-stabilization is that, if one of the x_p fields is dynamically changed, the protocol will effectively broadcast the new x_p value to other processors. Of particular interest are some properties that relate a sequence of changes to an x_p field to the sequence of x_p values observed at another processor q. More specifically, if processor p writes, over the course of a computation, the values $c_1, c_2, \ldots$ into x_p, and no processor q reads (via update images of x_p) a value c_k and then later reads a value c_j for $j < k$, then we call the update protocol *monotonic*. A monotonic update protocol guarantees that the sequence of values in any field image is a subsequence of the values written to the corresponding register field. Appendix B contains a proof showing that the protocol in Figure 3 is monotonic in any computation starting from a legitimate state. In the context of a dynamic system, we could also require that monotonicity hold in any trajectory that begins with a legitimate state. Unfortunately, the update protocol of Figure 3 does not have this stronger property[3]; however, a limited form called *impulse* monotonicity is satisfied.

6.1.1 Impulse Monotonicity

6.1.1-1 Let σ be a legitimate state for the update protocol in a topology $\mathcal{T}$ where $x_p = c_0$ at σ. Let λ be the state obtained by making a single topology change $\mathcal{E}$ to $\mathcal{T}$ and the assignment $x_p := c_1$. Let Φ be a topology-constant computation originating with state λ. Impulse monotonicity is satisfied if, for any states ρ and γ in Φ such that ρ occurs before γ: if processor q sees c_1 as the value of x_p at state ρ, then q sees c_1 as the value of x_p at state γ.

6.1.1-2 Impulse monotonicity is useful in the following way. If x_p is changed "slowly enough," meaning that the protocol successfully stabilizes between changes to x_p, then a FIFO broadcast of x_p values is obtained. In the sequel, we introduce an acknowledgment mechanism so that a processor does not change the broadcast value of interest until all other processors within a connected component have received the current value.

Corollary 1 *The protocol of Figure 3 enjoys impulse monotonicity.*

Proof of Corollary 1 The corollary is a specialization of Theorem 10, stated and proven in Appendix B.

Proof of Corollary 1 □

6.2 The Superstabilizer

6.2-1 The superstabilizer is a tool used in transforming a given self-stabilizing protocol P into a superstabilizing protocol P^s. Function $\mathcal{F}$, formally described

[3] To construct a monotonic update protocol, the update protocol can be modified by tagging x_p with a sequence number that increases with any x_p change; these sequence numbers are unbounded.

in Section 3, determines a new legitimate state from a previously legitimate state perturbed by a topology change $\mathcal{E}$. Our superstabilizer makes use of $\mathcal{F}$ by assembling the image of a global state σ from local snapshots, followed by disseminating $\mathcal{F}(\sigma)$ to all processors so that a new, legitimate global state can be installed. It is the responsibility of the superstabilizer to "hide" $\mathcal{E}$ from any processor in such a way that no user of protocol P can observe a state inconsistent with the current topology; this is done by making the global transition between legitimate states for different topologies effectively atomic, thus sparing protocol P from any stabilization effort.

6.2-2 The superstabilizer consists of two components: a modified version of the update protocol and an interrupt statement. By first modifying the given protocol P, and then adding the superstabilizer, we obtain the composite protocol P^s. We modify P as follows: each action of P at processor p is guarded by a Boolean variable freeze_p so that when freeze_p holds, no action of P is enabled at processor p and the program counter (with respect to P) remains static. Our superstabilizer will ensure that, starting from any initial state, all *freeze* fields eventually become *false* in the absence of topology changes, which then allows P to progress normally. The composition of P and the superstabilizer is the following for each processor p: while freeze_p holds, then the superstabilizer makes all steps at processor p; if freeze_p does not hold, then each read step in P is preceded by a complete cycle of the superstabilizer at processor p. In other words, the composition of the superstabilizer and P arranges step scheduling so that a cycle of the superstabilizer—from the first to the last step of the superstabilizer—is inserted before each read step of protocol P. This will ensure that any "news" of a topology change is processed by the superstabilizer before P at each processor. The superstabilizer also specifies an interrupt section for the composite protocol P^s, so that topology changes incident on processor p are handled by the superstabilizer at p.

6.3 The Superstabilizing Protocol

6.3-1 The combination of the superstabilizer and the modified protocol P results in a superstabilizing protocol P^s. A legitimate state for P^s is any state in which:

1. the variables, fields, and program counter with respect to P satisfy $\mathcal{LP}$ (where $\mathcal{LP}$ is the legitimacy predicate for the base protocol P);

2. the update protocol component of the superstabilizer is in a legitimate state (all e fields have accurate tuples);

3. every *freeze* variable is *false*; and

4. each computation that starts in such a state preserves points 1–3.

6.3-2 The passage predicate for our general method is defined in terms of the *freeze* fields. For any state σ, let $\mathit{warm}(\sigma)$ denote the set of processors having *freeze* fields that are *false*. Let $\sigma[\mathit{warm}]$ be the vector of local states of processors in $\mathit{warm}(\sigma)$. We call σ *warm-legitimate* if there exists a state γ (and a topology $\mathcal{T}.\gamma$) where $\gamma \vdash \mathcal{LP}$, such that $\sigma[\mathit{warm}] = \gamma[\mathit{warm}]$. In other words, σ is warm-legitimate if it appears to be a legitimate state (with respect to some topology) when we disregard any processor p with $\mathit{freeze}_p = \mathit{true}$. The passage predicate for the general method is: the protocol state is warm-legitimate.

6.3-3 The interface between the superstabilizer and P at processor p consists not only of the freeze_p variable, but a pseudo-variable snap_p, which is defined to be the collection, with respect to protocol P, of all local variables, shared fields, and the program counter of P for processor p. The superstabilizer can read and write snap_p. We denote by *snap* a set of snap_p variables, one for each processor. Our general method is, in brief, the following: after a topology change, P is frozen at all processors and a *snap* value is recorded; subsequently, a *snap* value appropriate for the new topology is computed, and each frozen processor is assigned its portion of the new *snap* value; and finally, all processors are thawed.

6.3-4 The programming notation given in Section 2 makes local images of register fields available to program operations: such images can be of a processor's own register, or that of its neighboring processors; for example, the code of Figure 3 permits processor p to refer to e_q for $q \in N_p$. The update protocol makes an image of each processor's x field available to every other processor within a connected component. For the superstabilizer, we extend the programming notation to allow any processor to refer to fields of any other processor. Thus processor p can refer to x_q for any $q \in \mathsf{processors}(e_p)$ by using images provided in the e field's tuples. Of course, these images may be out-of-date, which necessitates synchronization measures in the superstabilizer; such synchronization is achieved in *phases* to coordinate freezing and snapshots.

6.3-5 For convenience in describing the superstabilizer, we divide the x field into four subfields: $x_p \ = \ [\ a_p\ h_p\ t_p\ u_p\ ]$. To control the phases of superstabilization, the subfield a_p is used; it is a ternary-valued subfield provided for the three phases of superstabilization. These phases are as follows:

- *Phase 0* is the normal state of the superstabilizer, in which protocol P is active and the superstabilizer is idle. When $(\forall p ::\ a_p = 0)$ holds, we consider the superstabilization to be inactive (terminated).
- *Phase 1* consists of freezing protocol P and collecting snapshots from the frozen processors; also, in this phase an election takes place among all processors incident on a topology change to determine a single coordinator of the following phase. Phase 1 is active if $(\exists p ::\ a_p = 1)$ and $(\forall p ::\ a_p \leq 1)$.
- *Phase 2* is concerned with computing a new global state for protocol P and distributing the new state to all processors. Phase 2 is active if $(\exists p ::\ a_p = 2)$ remains active until acknowledged by all processors, and thereafter terminates in order to resume execution of phase 0.

6.3-6 To detect progress of phases, we employ an acknowledgment subfield h_p. This subfield is a vector of ternary values whose elements are images of other processor $a$subfields known to p: the protocol sets $h_p[r]$ to contain the image of a_r, as determined from p's image of x_r broadcast via the update protocol. Further, since h_p is broadcast via the update protocol to every processor, it is possible for a processor r to test the status of every other processor's image of a_r.

6.3-7 In addition to the a and h subfields, we define additional subfields of x_p to contain *snap* values. Subfield t_p contains a value of type snap_p, which is the portion of the state of p that is related to P. The subfield u_p contains a global *snap* value, i.e., $u_p[r]$ contains a snap_r value. Subfield u_p is used to broadcast a new global state. We denote by s_p the *snap* value that p assembles from the collection of t_r subfields obtained from x_r images. The assignment $u_p := \mathcal{F}(s_p)$ will determine a new legitimate state following a topology change (statement $\mathsf{U2}$).

6.3-8 To make a concise presentation, an additional device is used in the code of the interrupt statement. The function $\mathsf{refresh}(e_p)$ reproduces e_p except that the value of the x_p field is updated, i.e.,

$$\mathsf{refresh}(e_p) \ = \ (e_p \setminus\setminus \langle p, *, * \rangle) \ \cup \ \{\langle p, x_p, 0 \rangle\}$$

6.3-9 The interrupt statement for the superstabilizer is given in Figure 4. In response to a topology change $\mathcal{E}$ incident on processor p, the program counter (of protocol P^s) is reset to the first statement of the superstabilizer, the neighborhood N_p is adjusted to reflect $\mathcal{E}$, and the write operation is atomically executed. This operation halts P by setting freeze_p to *true*.

6.3-10 The remaining component of the superstabilizer consists of the combination of Figures 3 and 4, i.e., it is a modified update protocol. Statements U1–U8 are inserted between statements C3 and C4 to obtain the complete protocol. All quantifications over processors in expressions (such as $(\forall q :: \ a_q = 0)$) are implicitly quantified over $\mathsf{processors}(A) \cup \{p\}$ in the superstabilizer code.

6.3-11 The following scenario for a topology change $\mathcal{E}$ from a legitimate state illustrates the role of each statement of the superstabilizer. When statement E1 executes at each processor in the set S of nodes incident on $\mathcal{E}$, the base protocol P is frozen, a local snapshot is taken, and phase 1 of superstabilization begins. Then each process in S executes statements C1–C3 of the update protocol, but none of the statements U1–U8 make any variable assignments; statements C4–C6 of the update protocol ensure that a variables will be subsequently broadcast to indicate phase 1 is under way. Processors outside set S will take a snapshot at U8 when they detect that phase 1 is under way: this is the *freeze* wave. Each processor in S thus initiates its own *freeze* wave, and remains in phase 1 until either statement U1 or U2 assigns to the a variable. Statement U2 moves processor $p \in S$ from phase 1 to 2, but only after p has detected that every processor in the system has acknowledged that $a_p = 1$ (via the $h_q[p]$ subfields). Statement U1 causes a processor p to revert from phase 1 to 0 when some processor $q \in S$ with a larger identifier than p is detected—this is a form of leader election to insure that only one member of S persists in phase 1 (an additional conjunct in U2 forces the leader to wait until the election terminates before phase 2 begins). Let s be the leader of S, that is, s has the maximum identifier among processors in S. After statement U2 assigns $a_s := 2$ (and simultaneously computes a global legitimate state for the new topology), processor s broadcasts its a_s value via the update protocol and awaits acknowledgment of $a_s = 2$ via $h_q[s]$ subfields. When all processors have acknowledged $a_s = 2$, phase 2 is complete, and processor s executes the assignment in U3, whereby s makes the transition from phase 2 to phase 0. As $a_s = 0$ is broadcast throughout the system, statement U8 is executed by all processors to thaw the frozen system. Finally, statements U5–U7 are concerned with the acknowledgment of phases as seen by a-variable images and reflected by h-variable subfields.

The conditions for assignment in U5–U7 ensure the superstabilizer works properly in spite of the update protocol's weak property of monotonicity.

Theorem 6 *Protocol P^s is self-stabilizing with self-stabilization time $O(d+K)$, where K is the self-stabilization time of protocol P.*

Proof of Theorem 6 For simplicity, we assume the network is connected. By the construction of P^s, it suffices to prove that the superstabilizer converges in $O(d)$ rounds to $\mathcal{A} = (\forall q :: \ \neg \mathit{freeze}_q)$, and to show that $\mathcal{A}$ (or some stronger predicate) is stable. Thereafter, in $O(K)$ additional rounds, by base protocol P's self-stabilization, P^s stabilizes. Only statement U8 of the protocol assigns to freeze_p; if we show P^s stabilizes to $(\forall q :: a_q = 0)$, then by Theorem 5, all a-field images are broadcast in $O(d)$ rounds; all *freeze* variables are *false* in the following round. Notice that $a_p = 0$ is stable for any p, since none of U1–U8 assign to a_p if $a_p = 0$ holds. Therefore it suffices to show that *some* state satisfying $a_q = 0$ occurs for each processor q within $O(d)$ rounds of any computation. Heading for a contradiction, let processor r be a processor such that $a_r \neq 0$ holds for more than $O(d)$ rounds of some computation; because none of U1–U8 assign $a_r := 1$, yet $a_r = 1$ is the precondition of assigning $a_r := 2$ (see U2), we deduce that $a_r \neq 0$ holds continuously for more than $O(d)$ rounds. Suppose $a_r = 2$ holds continuously; by acknowledgments from U5–U7 and stabilization of the update protocol, U3 eventually assigns $a_r := 0$, which is a contradiction. It remains to consider that $a_r = 1$. There may be more than one processor for which the a field is continuously 1-valued; let t be such a processor of maximum identifier. By acknowledgments U5–U7 and the update protocol, any processor $q \neq t$ having $a_q = 1$ assigns $a_q := 0$ at U1 within $O(d)$ rounds (because $q < t$). Thus, after $O(d)$ rounds, t is the only processor having a 1-valued a field. But again, by acknowledgments U5–U7 and the update protocol, t subsequently assigns $a_r := 2$ at U2 within $O(d)$ rounds, which leads to the contradiction described above.

Proof of Theorem 6 □

Theorem 7 *Protocol P^s is superstabilizing with superstabilization time $O(d)$.*

Proof of Theorem 7 Consider a computation beginning from a state σ that is the result of a single topology change event $\mathcal{E}$ at a legitimate state. Our obligation is to show that the passage predicate holds until a legitimate state is obtained (recall that the passage predicate for P^s is that the state be warm-legitimate). By $\mathsf{E1}$, $\sigma \vdash a_r = 1 \wedge \mathit{freeze}_r$ holds for every r incident on $\mathcal{E}$; thus σ is warm-legitimate. As the update protocol broadcasts the 1-valued a fields, statement $\mathsf{U8}$ sets freeze_p at all processors p within $O(d)$ rounds. Statement $\mathsf{U1}$ only assigns $a_p := 0$ for $p < q$, where q has a larger identifier, and all processors have observed $a_q = 1$. Therefore, within each connected component of the network, the processor with maximum identifier incident on $\mathcal{E}$—which we call the "leader processor" t of the component—does not execute the assignment of $\mathsf{U1}$. Thanks to impulse monotonicity, the condition $(\exists q :: \ a_q \neq 0)$ observed by update images is stable so long as processor t does not change its 1-valued a_t field. Within $O(d)$ rounds, statement $\mathsf{U1}$ assures that only one processor t (per connected component) satisfies $a_t = 1$; in at most $O(d)$ subsequent rounds, by acknowledgments and the update protocol, statement $\mathsf{U2}$ executes $a_t := 2$ at processor t. At this point, we assert that all processors are frozen; note also that each global state from σ to this point is warm-legitimate. Upon execution of $\mathsf{U2}$, a new global state is computed from combined snapshots. Although $a_t = 2$ is not broadcast monotonically, statements $\mathsf{U6}$ and $\mathsf{U7}$ are coded in such a way that acknowledgment of $a_t = 2$ is monotonic. Therefore, once t observes $(\forall q :: \ h_q[p] \neq 1)$, it is the case that every processor has received the new global state and assigned P's fields and variables. Thus after $O(d)$ rounds, t executes $\mathsf{U3}$, and the final phase of the superstabilizer begins. In this final phase, stabilization to $(\forall q :: \ \neg\mathit{freeze}_q)$ occurs within $O(d)$ rounds; the passage predicate holds because at each state, the subset of unfrozen processors is locally legitimate for the new topology.

Proof of Theorem 7 □

7 Conclusions

7-1 There is increasing recognition that dynamic protocols are necessary for many networks. Studying different approaches to programming for dynamic environments is therefore a motivated research topic. Although self-stabilizing techniques for dynamic systems have been previously suggested, explicit research to show how and where these techniques are useful has been lacking. This paper shows how assumptions about interrupts and dynamic change can be exploited with qualitative and quantitative advantages, while retaining the fault-tolerant properties of self-stabilization.

```
Superstabilization Section:
        (statements C1–C3 of update protocol)
        U1    if $(a_p = 1 \wedge (\exists q :: a_q \neq 0 \wedge q > p \wedge (\forall r :: h_r[q] \neq 0)))$
                  then $a_p := 0$
        U2    if $(a_p = 1 \wedge (\forall q : q \neq p : a_q = 0) \wedge (\forall q :: h_q[p] \neq 0))$
                  then $a_p, u_p := 2, \mathcal{F}(s_p)$
        U3    if $(a_p = 2 \wedge (\forall q :: h_q[p] \neq 1))$
                  then $a_p := 0$
        U4    forall $q \in \mathsf{processors}(A) \cup \{p\}$
                  do
        U5        if $a_q = 0$ then $h_p[q] := 0$
        U6        if $a_q = 1 \wedge h_p[q] = 0$ then $h_p[q] := 1$
        U7        if $a_q = 2 \wedge h_p[q] = 1$ then $h_p[q], \mathsf{snap}_p := 2, u_q[p]$
                  od
        U8    if $(\exists q \in \mathsf{processors}(A) \cup \{p\} :: a_q \neq 0)$
                  then $\mathit{freeze}_p , t_p := \mathit{true}, \mathsf{snap}_p$
                  else $\mathit{freeze}_p := \mathit{false}$
        (statements C4–C6 of update protocol)
Interrupt Section:
        E1    write
                  $\left( \begin{array}{lll} a_p & := & 1 \\ \mathit{freeze}_p & := & \mathit{true} \\ t_p & := & \mathsf{snap}_p \\ e_p & := & \begin{cases} \emptyset & \text{if } \mathcal{E} = \mathsf{recov}_p \\ \mathsf{refresh}(e_p) & \text{if } \mathcal{E} \neq \mathsf{recov}_p \end{cases} \end{array} \right)$
```

Figure 4: Superstabilizer: update extension and interrupt for p

7-2 In particular, we suggest that when the system is in an illegitimate configuration, reset to a predefined configuration will not take place; instead, the system will reach a legitimate configuration that is close to the current illegitimate configuration (where “close” means a small adjustment measure). The benefits of this approach are twofold. First, such a strategy may keep most of the sites of the system unchanged and in working order (as in the example of connections within an unchanged portion of a spanning tree). Second, in some cases the amount of work (superstabilizing time) required to reach a close legitimate state can be small (as in our coloring example).

Acknowledgment

We thank Ajoy Kumar Datta and an anonymous referee for helpful remarks.

Appendix A: Spanning Tree Proofs

Theorem 3 *The spanning tree protocol self-stabilizes in $O(n)$ rounds.*

Proof of Theorem 3 Proof by induction on an arbitrary computation Φ. The induction is based on a directed tree. Let T_r be the maximum subset of processors whereby:

1. $d_r = 0$,
2. the set $\{t_p \mid p \in T_r \ \wedge \ p \neq r\}$ represents a directed tree rooted at r,
3. for $p \in T_r$ and $p \neq r$, register field d_p satisfies $d_p = 1 + d_q$ where $q = t_p$, and
4. each processor in T_r has executed at least one cycle in Φ.

After one round, $d_r = 0$ holds for the remainder of the computation. Therefore, after the first round, T_r is nonempty, containing at least r. The remainder of the proof concerns rounds two and higher, and is organized into the following three claims.

> **Claim 1 ($\boldsymbol{T_r}$ is Stable)** *If $p \in T_r$ holds at the beginning of the round, then t_p and d_p do not change during the round. The claim follows by induction on depth of the tree T_r. First we strengthen*

the hypothesis to state that any node $p \in T_r$ satisfies $d_r = k$, where k is the depth of p in T_r. The basis for the induction is the root r, which satisfies $d_r = 0$; statement S4 assures that d_r does not change during the round. Suppose all nodes of T_r up to depth k satisfy the hypothesis, i.e., they are stable and have a d field equal to the node depth. Consider some $v \in T_r$ at depth $k+1$; by (2) and (3), $d_v = k+1$ at the beginning of the round. Since every d field has non-negative value and $w_{vz} = n$ for any $z \neq t_v$, and assuming the inductive hypothesis for the node named by t_v, statement S3 cannot compute any lower value for x than $k+1$, and the values t_v and d_v do not change during the round.

Claim 2 (T_r Growth) *If there exists a processor that is not contained in T_r and $(\forall p : \ p \notin T_r : \ d_p > 2n)$ holds at the beginning of the round, then T_r grows by at least one processor by the end of the round. The claim follows by examining processors outside of T_r that also neighbor T_r. Let p be such a processor, outside T_r and neighbor to $q \in T_r$. By Claim 1, $d_q + w_{pq} < 2n$. Therefore, during the round, p cannot choose t_p to be some processor s satisfying $d_s > 2n$. Thus T_r grows by at least one processor.*

Claim 3 (d_p Growth) *Define M_i to be the minimum d-register value of any processor outside of T_r in round i; then $M_{i+1} > M_i$. The claim is verified by considering, for round i and $p \notin T_r$, assignment to each d_p register in that round. During a round, the value obtained for d_p is strictly larger than that of some neighboring d_q; if $q \in T_r$, then $p \in T_r$ holds at the end of the round; and if $q \notin T_r$, then the claim holds.*

A corollary of Claim 3 is that following rounds $2n+2$ and higher, for every $p \notin T_r$, the field d_p satisfies $d_p > 2n$. Consequently, for rounds $2n+2$ and higher, by Claim 2, if T_r does not contain all processors, then T_r grows by at least one processor in each successive round. The theorem follows because there are at most n processors.

Proof of Theorem 3 □

Theorem 4 *The spanning tree protocol is superstabilizing for the class Λ, with superstabilization time $O(1)$ and adjustment measure $\mathcal{R} = 1$.*

Proof of Theorem 4 We show that starting from a state δ, $\delta \vdash \mathcal{L}$, followed by a topology change $\mathcal{E}$, $\mathcal{E} \in \Lambda$, resulting in a state σ, that $\sigma \vdash \mathcal{L}$ holds. In the case of $\mathcal{E} = \mathsf{crash}_{pq}$, removing a nontree link, for either processor p or q, the weight of the p-q link $w_{pq} = n$ at state δ. By assumption of $\delta \vdash \mathcal{L}$, it follows that computation of d and t fields produces identical results in any round following σ, since these are necessarily based on unit w values. In the case of $\mathcal{E} = \mathsf{recov}_{pq}$, the weight of the new p-q link is $w_{pq} = n$ at state σ; hence distances are not reduced by addition of the new link, and computation of d and t fields produces identical results in any round following σ. Therefore $\sigma \vdash \mathcal{L}$.

Proof of Theorem 4 □

Appendix B: Update Protocol

B-1 In addition to showing that the update protocol presented in Section 6 is self-stabilizing, additional theorems given in this appendix show that the update protocol enjoys some other properties related to broadcast monotonicity and the protocol's use of memory. We begin with the proof of self-stabilization.

B.1 Self-Stabilization

B.1-1

Theorem 5 *The update protocol of Figure 3 self-stabilizes in $O(d)$ rounds.*

Proof of Theorem 5 The proof is organized as three claims.

> **Claim 4** *Following round i, $i \geq 1$, the e_p field of every processor satisfies*
>
> $$(\forall p, q, j : \ \ j < i : \ \ dist(p,q) \leq j \ \ \Leftrightarrow \ \ (\exists \langle q, x_q, dist(p,q) \rangle \in e_p\,))$$
>
> *The claim follows by induction on i. The basis of the induction is the first round, which trivially establishes $\langle p, x_p, 0 \rangle \in e_p$ for every processor. The induction step follows because field e_p is assigned anew in each round and based on tuples that, by the induction hypothesis, have the required property.*

Claim 5 *Following round i, $i \geq 1$, the e_p field of every processor satisfies*

$$(\forall p, q, j : \ \ j < i : \ \ (\exists \langle q, y, k \rangle \in e_p :: \ \ k \leq j \ \Rightarrow \ (dist(p,q) = k \wedge y = x_q) \ ))$$

This claim follows by the same inductive argument presented for Claim 4.

Claim 6 *Following round $d+1$, $(\forall p :: \ \ (\forall \langle , , k \rangle \in e_p :: \ \ k \leq d \ ))$. The claim is shown by contradiction. Suppose e_p contains a tuple $\langle , , j \rangle$ where $j > d$. Observe that if $j > d+1$, then by the construction of the* initseq *function, at the end of round $d+1$ the field e_p also contains some tuple $\langle q, , k \rangle$ where $k = d+1$. Thus to show the claim, it suffices to show a contradiction for $k = d+1$. Since p assigned the tuple $\langle q, , d+1 \rangle$ to e_p during round $d+1$, it must be that p found at some neighbor s the tuple $\langle q, , d \rangle$, and found no tuple with q as the first component at a smaller distance. However, the tuple located at s having distance d represents the shortest distance to q by Claim 5. And since d bounds the maximum possible shortest path, by Claim 4 all shortest paths between p and q are visible to p at the end of round d. We conclude that $dist(p,q) = d+1$, which contradicts the definition of diameter d.*

Claims 4–6 together imply that, following $d+1$ rounds, each processor correctly has a tuple for every other processor at distance d, and every tuple in an e field correctly refers to a processor.

Proof of Theorem 5 □

B.2 Memory Adaptivity

B.2-1 Nowhere in the code of the update protocol is the size of the network used, nor is a bound on the number of processors in a connected component assumed; consequently, any number of processors can be dynamically added to the system, provided processor identifiers are unique. Moreover, the local implementation of operations on processor variables A, B, and even the field e_p can use dynamic memory allocation. The following lemma shows that dynamic memory operations do not use unbounded amounts of memory.

Lemma 1 *For any computation Φ of the update protocol, no processor requires more than $O(\Delta \cdot K \cdot n)$ space for variables and register fields, where in the initial state of Φ: Δ is the maximum number of neighbors a processor has, n is the number of processors, and K is the maximum number of tuples of any processor's A, B, or e field in the initial state of Φ.*

Proof of Lemma 1 The computation of B consists of at most nK tuples, since tuples with duplicate identifiers are not added to B by $\mathsf{C4}$, and the number of identifiers is bounded by nK. Moreover, no statement is capable of introducing a tuple with a processor identifier not already present in another tuple. Hence any assignment by $\mathsf{C5}$ places at most nK tuples in e_p. The collection procedure for constructing A is the union of at most Δ sets of at most nK tuples (K tuples in the first round, and nK tuples in subsequent rounds).

Proof of Lemma 1 □

Although the lemma shows that the update protocol does not use unbounded space in its computation, this is insufficient for a self-stabilizing implementation: suppose processors are implemented on machines with fixed memory limits and an initial state of a computation is such that the number of tuples is at or near the memory limit; subsequent computation may then abort by exceeding the memory limit in a dynamic allocation request. Therefore, in order to claim that the update protocol is self-stabilizing, we assume that every trajectory's initial state satisfies $nK \leq \mathcal{N}$, where $\mathcal{N}$ is some appropriate limit related to memory limits of processors (even if the abort resets memory, some minimal amount of memory is needed to guarantee self-stabilization of the update protocol).

B.2-2 Note that upon stabilization, the e_p register contains only those tuples representing reachable nodes in the network. Therefore, the amount of memory needed for e_p can be dynamically adjusted during a computation to the minimum amount needed to represent the list of tuples; this idea is called *memory adaptivity* in [AEYH92]. The following lemma is an observation due to Gerard Tel.[4]

Lemma 2 *The update protocol of Figure 3 is memory adaptive.*

[4] Remark during presentation, December 1993.

Proof of Lemma 2 Upon stabilization, the necessary size of the e field is bounded by a function of the number of processors.

Proof of Lemma 2 □

B.3 Monotonicity Properties

B.3-1 We distinguish three *monotonicity* properties of an update protocol:

- *Static monotonicity.* Let σ be a legitimate state for the update protocol where $x_p = c_0$ at σ. Suppose Φ is a topology-constant computation originating with state σ, and suppose field x_p is changed at distinct states $\delta_1, \delta_2, \ldots$ of Φ to have the values $c_1, c_2, \ldots$, where state δ_i occurs before δ_j for $i < j$. Static monotonicity is satisfied if, for any states ρ and γ in Φ such that ρ occurs before γ: if processor q sees c_i as the value of x_p at state ρ and sees c_j as the value of x_p at state γ, then $i \leq j$ holds.

- *Dynamic monotonicity.* Let σ be a legitimate state for the update protocol where $x_p = c_0$ at σ. Suppose Φ is a trajectory originating with state σ, and suppose field x_p is changed at distinct states $\delta_1, \delta_2, \ldots$ of Φ to have the values $c_1, c_2, \ldots$, where state δ_i occurs before δ_j for $i < j$; Φ may have topology changes interleaved with steps of processors, including, possibly, the crash and recovery of processor p. Dynamic monotonicity is satisfied if, for any states ρ and γ in Φ such that ρ occurs before γ: if processor q sees c_i as the value of x_p at state ρ and sees c_j as the value of x_p at state γ, then $i \leq j$ holds.

- *Impulse monotonicity.* Let σ be a legitimate state for the update protocol in a topology $\mathcal{T}$ where $x_p = c_0$ at σ. Let λ be the state obtained by making a single topology change $\mathcal{E}$ to $\mathcal{T}$ and the assignment $x_p := c_1$. Let Φ be a topology-constant computation originating with state λ. Impulse monotonicity is satisfied if, for any states ρ and γ in Φ such that ρ occurs before γ: if processor q sees c_1 as the value of x_p at state ρ, then q sees c_1 as the value of x_p at state γ.

Note that with static and dynamic monotonicity, we admit the possibility of "overwriting" of x_p before its value is successfully broadcast to all processors; however, a subsequence of FIFO delivery is guaranteed by monotonicity. If

x_p is changed "slowly enough," meaning that the protocol successfully stabilizes between changes to x_p, then a FIFO broadcast of x_p values is obtained. In Section 6, we introduce an acknowledgment mechanism so that a processor does not change the broadcast value of interest until all other processors within a connected component have received the current value. The acknowledgment mechanism does not itself guarantee FIFO broadcast; monotonicity is also required. As indicated in following theorems, the update protocol satisfies static and impulse monotonicity, but not dynamic monotonicity; further measures are introduced in Section 6 to deal with the lack of dynamic monotonicity.

Theorem 8 *The update protocol of Figure 3 enjoys static monotonicity.*

Proof of Theorem 8 The theorem can be proved by induction on a lexicographic measure composed of path length and the ordering of links by a processor's neighborhood; essentially, the deterministic ordering of links defines a broadcast tree. Our general method does not exploit static monotonicity, so we omit details of the proof.

Proof of Theorem 8 □

B.3-4 In the sequel, for dynamic and impulse monotonicity, we make a restriction on a topology change event $\mathcal{E}$ that adds a node p to the network: the e_p field contains no tuples. Given this restriction, the following monotonicity theorems hold.

Theorem 9 *The update protocol of Figure 3 does not satisfy dynamic monotonicity.*

A counter-example provides proof of this theorem, and is presented in [DH95]. The counter-example actually shows that the update protocol does not satisfy even more restricted forms of dynamic monotonicity: the example is constructed with a single initial topology change and no further topology changes, and only two changes to a register field.

Theorem 10 *The protocol of Figure 3 is impulse monotonic:*

- *Let σ be a legitimate state for topology $\mathcal{T}$.*
- *The following atomically occurs at state σ: a single topology change $\mathcal{E}$ occurs to obtain a new topology $\mathcal{U}$, and for each processor r incident on $\mathcal{E}$, the value of x_r is changed to $\ddot{x}_r$.*

- *Let Φ be a topology-constant computation of the protocol following the change $\mathcal{E}$.*

Then the following two claims hold:

- *For any two processors p and q that are connected in both $\mathcal{T}$ and $\mathcal{U}$, there is a tuple $\langle p,,\rangle \in e_q$ at each state in Φ; if p and q are not connected in $\mathcal{T}$ but are connected in $\mathcal{U}$, then for any state $\delta \in \Phi$ satisfying $\langle p,,\rangle \in e_q$, at every subsequent state ρ there is a tuple $\langle p,,\rangle \in e_q$.*
- *For any processors p and q, where p is incident on $\mathcal{E}$, if there is a tuple $\langle p, \ddot{x}_p, \rangle \in e_q$ at some state δ, then at every state ρ following δ in Φ there is a tuple $\langle p, \ddot{x}_p, \rangle \in e_q$.*

Proof of Theorem 10 The proof is based on considering an arbitrary legitimate state σ in topology $\mathcal{T}$, an arbitrary single topology change $\mathcal{E}$ in state σ resulting in topology $\mathcal{U}$, followed by a topology-constant computation Φ. We consider two cases based on how $\mathcal{E}$ changes: either $\mathcal{E}$ increases or decreases connectivity in the network. We label a topology change that increases connectivity as $+\mathcal{E}$, since either a link or a processor is added to the network; a topology change that decreases connectivity is labeled $-\mathcal{E}$.

Proof of Theorem 10-2

For the $+\mathcal{E}$ case, a technical lemma is needed: Lemma 4 shows for Φ that distances tracked in tuples do not increase during the computation, and that the function initseq does not remove tuples during the course of the protocol's computation. To show impulse monotonicity, we assign one of two colors to each tuple in an e register. Atomically, with $+\mathcal{E}$ we color all tuples white with the exception of $\langle,,0\rangle$ tuples incident on $+\mathcal{E}$, which are colored black. Then at each cycle of a processor in Φ, the color of a $\langle,,0\rangle$ tuple is black for processors incident on $+\mathcal{E}$, and white for other processors; the color of a $\langle,,k\rangle$ tuple, $k \neq 0$, is inherited from the color of the $\langle,,(k-1)\rangle$ tuple upon which it is based. It follows that any tuple that decreases distance during the course of Φ is black; because distances do not increase in Φ and the ordering of neighbors is deterministic in the protocol, once a tuple is black it remains black. Thus for an arbitrary processor q and some p incident on $+\mathcal{E}$, the tuple $\langle p,,\rangle \in q$ changes color exactly once in computation Φ.

Proof of Theorem 10-3

For the $-\mathcal{E}$ case, the same coloring technique is used, with a different lemma: Lemma 3 shows for Φ that distances tracked in tuples do not decrease during the computation and that initseq does not remove tuples that refer to reachable processors. To show impulse monotonicity, we assign one of two colors to each tuple in an e register. Atomically, with $-\mathcal{E}$ we color all tuples white with the exception of $\langle,,0\rangle$ tuples incident on $-\mathcal{E}$, which are colored black. Then at each cycle of a processor in Φ, the color of a $\langle,,0\rangle$ tuple is black for processors incident on $-\mathcal{E}$, and white for other processors; the color of a $\langle,,k\rangle$ tuple, $k \neq 0$, is inherited from the color of the $\langle,,(k-1)\rangle$ tuple upon which it is based. It follows that any tuple that increases distance during the course of Φ is white unless it represents a final increase basing the tuple on a shortest path for $\mathcal{U}$; because distances do not decrease in Φ and the ordering of neighbors is deterministic in the protocol, once a tuple is black it remains black. Thus for an arbitrary processor q and some p incident on $-\mathcal{E}$, the tuple $\langle p,,\rangle \in q$ changes color exactly once in computation Φ.

Proof of Theorem 10 □

B.3-5 For the remaining lemmas of this subsection, σ, $\mathcal{E}$, $\mathcal{T}$, $\mathcal{U}$, and Φ are fixed as specified in Theorem 10. Let $dist(x,y)=\infty$ denote that no path connects x and y. To simplify analysis, we call a tuple $\langle p,,\rangle \in e_q$ a *reachable* tuple if $dist_{\mathcal{U}}(p,q) \neq \infty$.

B.3-6 Let ρ and δ be states of Φ. The notation $\rho \prec \delta$ denotes that ρ occurs before δ in the sequence Φ. The notation $successor(\rho)=\delta$ means that state δ immediately follows ρ in Φ. The notation $\langle p,,k\rangle \in e_q \odot \delta$ means that tuple $\langle p,,k\rangle$ is contained in field e_q at state δ. The predicate $adjust(q,\rho,\delta)$ is defined to hold if a distance change in a reachable tuple occurs:

$$adjust(q,\rho,\delta) \;\equiv\; \delta = successor(\rho) \wedge (\exists \langle p,,k\rangle \in e_q {\odot} \rho :: \; (\exists \langle p,,m\rangle \in e_q {\odot} \delta :: \; m \neq k))$$

We define a *based tuple* recursively as follows: tuple $\langle p,,k\rangle \in e_q \odot \rho$ is *based* if $k=0$ or there is some based tuple $\langle p,,(k-1)\rangle \in e_r \odot \rho$ for $r \in N_q$. Observe that in a legitimate state for the update protocol, all tuples are based; following event $-\mathcal{E}$, some tuple(s) may not be based.

b.3-7 A tuple $\langle p,,k\rangle \in e_q$ is *low* if it is reachable and $k < dist_{\mathcal{U}}(p,q)$. A tuple $\langle p,,k\rangle \in e_q$ is said to be *maxlow* if it is low and satisfies:

$$(\forall \langle s,,m\rangle \in e_q :: \; \langle s,,m\rangle \text{ is low} \;\Rightarrow\; dist_{\mathcal{U}}(s,q) \leq dist_{\mathcal{U}}(p,q))$$

Lemma 3 *For event $-\mathcal{E}$, for all processors p and q satisfying $dist_{\mathcal{U}}(p,q) \neq \infty$, the following claims hold:*

Claim 7 $(\forall \rho : \; \rho \in \Phi : \; \langle p,,\rangle \in e_q \odot \rho)$

Claim 8 $\langle p, , \ell\rangle \in e_q \;\Rightarrow\; \ell \leq \mathit{dist}_{\mathcal{U}}(p, q)$

Claim 9 $(\rho \prec \delta \wedge \langle p, , \ell\rangle \in e_q \odot \rho \wedge \langle p, , m\rangle \in e_q \odot \delta) \;\Rightarrow\; \ell \leq m$

Claim 10 $\langle p, , k\rangle \in e_q \odot \rho$ *is based* $\Rightarrow\; \mathit{dist}_{\mathcal{U}}(q, p) = k$

Claim 11 $\langle p, , k\rangle \in e_q$ *is based* $\Rightarrow (\forall j : 0 \leq j \leq k : (\exists \langle r, , j\rangle \in e_q :: \langle r, , j\rangle$ *is based*$))$

Claim 12 $\mathit{adjust}(q, \rho, \delta) \;\Rightarrow\; (\forall \langle s, , \rangle \in e_q :: \langle s, , \rangle \in e_q \odot \rho$ *is maxlow* $\Rightarrow \langle s, , \rangle \in e_q \odot \delta$ *is based*$)$

Proof of Lemma 3 Proof is by induction on Φ.

Proof of Lemma 3-1 **Basis** Let λ be the state obtained from σ as modified by $-\mathcal{E}$; λ is the initial state of Φ and forms the induction's basis. Claim 7 holds for λ by the assumption that σ satisfies $\mathcal{L}_{\mathcal{T}}$. Claims 8 and 10 hold by the assumption of σ satisfying $\mathcal{L}_{\mathcal{T}}$ and the fact that $-\mathcal{E}$ can only increase minimum distances between processors. Claims 9 and 12 are claims over pairs of distinct states, and thus hold trivially in the initial state of Φ. Claim 11 follows from Claim 10, which establishes that a based tuple represents a minimum distance, and because σ satisfies $\mathcal{L}_{\mathcal{T}}$, each based tuple also corresponds to a minimum distance in $\mathcal{T}$; therefore, all nodes that lie on a shortest path unaffected by $-\mathcal{E}$ between q and p have based tuples.

Proof of Lemma 3-2 **Induction** Let $\delta = \mathit{successor}(\rho)$, and suppose that Claims 7–12 hold for all states γ, $\gamma \preceq \rho$. Consider two cases for *adjust*: if $\mathit{adjust}(q, \rho, \delta)$ does not hold for any processor q, that is, either e_q is unchanged by the transition from ρ to δ or only changes to unreachable tuples occur, then Claims 7–12 hold for δ by inheritance from ρ. The other possibility is that $\mathit{adjust}(q, \rho, \delta)$ holds for some processor q. In this case, the transition from ρ to δ writes $\mathsf{initseq}(B)$ into e_q, where B contains tuples computed by steps in Φ or that are present in state λ. Tuples placed in B by steps of Φ are calculated from tuples of q's neighbors, which satisfy Claims 7–12 by the induction hypothesis. To show that Claims 7–12 hold for state δ, we consider the claims with respect to B, and then reason about $\mathsf{initseq}(B)$. The remainder of the induction considers tuples placed in B by steps of Φ preceding state δ.

Proof of Lemma 3-3 Claim 7 holds for B because $\langle q, , 0\rangle \in e_q$ holds for any iteration of the loop in Figure 3 and by the induction hypothesis for Claim 7, each $r \in N_q$

has a tuple $\langle p,,\rangle \in e_r$ for any p satisfying $dist_{\mathcal{U}}(r,p) \neq \infty$. Claim 8 holds for B since $\langle p,,\ell\rangle \in B$ for $\ell \neq 0$ implies $\langle p,,\ell-1\rangle \in e_r$ for some $r \in N_q$, and the induction hypothesis for Claim 8 is assumed for r. Claim 8, the induction hypothesis of Claim 10, and the definition of a based tuple show that Claim 10 holds for based tuples in B. Claim 11 holds by the induction hypothesis on Claim 11: based tuples in B are calculated upon neighboring processor-based tuples, which satisfy Claim 11 by assumption; hence all of the neighbor's supporting based tuples (at smaller distances) are also input to forming tuples in B. Claims 9 and 12 are only concerned with tuples that change distance with respect to current distances in the e_q field. Claim 9 holds for tuples in B by the induction hypothesis for Claims 7 and 9; tuples in B are calculated from neighboring e fields and tuples in these fields do not increase distance by any transition prior to state δ. Similarly, Claim 12 holds for B, because any adjustment to a tuple follows from (possibly multiple) changes in neighboring e fields; by hypothesis 12, each such change to an e field adjusts all maxlow tuples, which then by Claims 7, 8, and 10 remain constant thereafter.

Proof of Lemma 3-4 Thus Claims 7–12 have been established for B prior to the writing of $\mathsf{initseq}(B)$ at state δ. It only remains to show that no reachable tuple is removed from B by the application of $\mathsf{initseq}$. This is argued by contradiction. Suppose a reachable tuple $\langle p,,m\rangle \in B$ is discarded by $\mathsf{initseq}$; this implies the existence of a "gap," i.e., for some distance ℓ, $\ell < m$, no tuple $\langle,,\ell\rangle \in B$ exists. All tuples contained in B have distances equal to or larger than tuples contained in e_q, by Claim 9. It follows that such a gap is the result of increasing the distance of some tuple(s). Yet Claim 12 implies that the maximum-distance reachable tuple resulting from an increase yields a based tuple; Claim 11 then implies the existence of tuples at all lesser distances in B, which contradicts the assumption of a gap.

Proof of Lemma 3 □

Lemma 4 *For event $+\mathcal{E}$, for all processors p and q satisfying $dist_{\mathcal{U}}(p,q) \neq \infty$, the following claims hold:*

Claim 13 $\langle p,,k\rangle \in e_q \;\Rightarrow\; dist_{\mathcal{U}}(p,q) \leq k$

Claim 14 $(\rho \prec \delta \wedge \langle p,,\ell\rangle \in e_q{\odot}\rho \wedge \langle p,,m\rangle \in e_q{\odot}\delta) \;\Rightarrow\; \ell \geq m$

Claim 15 $(\rho \prec \delta \;\wedge\; \langle p,,\rangle \in e_q \odot \rho) \;\Rightarrow\; \langle p,,\rangle \in e_q \odot \delta$

Claim 16 $\langle p,,k\rangle \in e_q \;\Rightarrow\; (\forall r,\ell :: \; \mathit{dist}_{\mathcal{T}}(p,r) = \ell \neq \infty \;\Rightarrow\; (\exists\langle r,,m\rangle \in e_q :: \; m \leq k+\ell))$

Proof of Lemma 4 Proof is by induction on Φ. To simplify cases within the proof, we distinguish two possibilities for event $+\mathcal{E}$; either a link is added to the network, or a node is added with its accompanying links. In case $+\mathcal{E}$ adds a node to the network, let z denote the node added. The assumption for dynamic and impulse monotonicity with respect to nodes is that they initially have empty e fields when added to the network. Observe from the code of the update protocol and the assumption of a legitimate state prior to $+\mathcal{E}$ that no processor changes its e field so long as e_z contains no tuples. Furthermore, after one cycle by processor z, the e_z field is assigned to satisfy:

(†) $(\forall p,k :: \; \mathit{dist}_{\mathcal{U}}(p,z) = k \neq \infty \;\Rightarrow\; \langle p,,k\rangle \in e_z)$

In addition to Claims 13–16, we add Claim 17 to the list of claims to prove invariant in the computation Φ:

Claim 17 $\langle z,,k\rangle \in e_q \;\Rightarrow\; (\forall r,\ell :: \; \mathit{dist}_{\mathcal{U}}(z,r) = \ell \neq \infty \;\Rightarrow\; (\exists\langle r,,m\rangle \in e_q :: \; m \leq k+\ell))$

Proof of Lemma 4-2

Basis If $+\mathcal{E}$ adds no processor to the network, then let λ be the state obtained from σ as modified by $+\mathcal{E}$; if a processor z is added to the network, then let λ be the first state in Φ that satisfies (†). State λ forms the induction's basis. Claim 13 holds for λ because event $+\mathcal{E}$ can only decrease distances between existing nodes, and all tuples present in e fields at state σ represent distances in $\mathcal{T}$ by the assumption of a legitimate state, hence also for state λ; and (†) directly implies Claim 13 for processor z. Claims 14 and 15 hold for λ either because there are no previous states in Φ or because no e fields are modified except for e_z, which obtains its initial value at λ. Claim 16 holds trivially for λ since σ satisfies $\mathcal{L}_{\mathcal{T}}$, and Claim 17 holds because no processor reads any tuple from e_z prior to state λ.

Proof of Lemma 4-3

Induction Let $\delta = \mathit{successor}(\rho)$, and suppose that Claims 13–17 hold for all states γ, $\gamma \preceq \rho$. Consider two cases for *adjust*: if $\mathit{adjust}(q,\rho,\delta)$ does not hold for any processor q, that is, either e_q is unchanged by the transition from ρ to δ or only changes to unreachable tuples occur, then Claims 13–17 hold for δ by inheritance from ρ. The other possibility is that $\mathit{adjust}(q,\rho,\delta)$ holds for some processor q. In this case, the transition from ρ to δ writes $\mathsf{initseq}(B)$ into e_q, where B contains tuples computed by steps in Φ or that are present in state λ. Tuples placed in B by steps of Φ are calculated from tuples of q's neighbors, which satisfy Claims 13–17 by the induction hypothesis. To show that Claims 13–17 hold for state δ, we consider the claims with respect to B, and then reason about $\mathsf{initseq}(B)$. The remainder of the induction considers tuples placed in B by steps of Φ preceding state δ.

Proof of Lemma 4-4 Claim 13 holds for B because any step of Φ that places a tuple in B either places $\langle q, , 0\rangle$ in B or calculates some $\langle p, , (k+1)\rangle$ based on a tuple $\langle p, , k\rangle \in e_r$ for some $r \in N_q$; and tuples in e_r satisfy Claim 13 by the induction hypothesis. Similarly, Claims 16 and 17 follow by appealing to the induction hypothesis for the contents of some neighboring processor's e field. To show Claim 15, consider any tuple $\langle p, , \rangle \in e_q \odot \rho$. This tuple's presence is either inherited from σ or was calculated by some step of Φ preceding δ; in either case, we infer the existence of a tuple $\langle p, , \rangle \in e_r$ for $r \in N_q$. By the induction hypothesis on Claim 15, some tuple $\langle p, , \rangle \in e_r$ is present at each state up to ρ, which implies the computation of B results in $\langle p, , \rangle \in B \odot \rho$. For Claim 14 it suffices to show, for any tuple $\langle p, , \ell\rangle \in e_q \odot \rho$, that $\langle p, , m\rangle \in B \odot \rho$ satisfies $m \leq \ell$. Since calculation of $\langle p, , m\rangle$ is based on neighboring e fields, all of whose tuples satisfy Claim 14 by hypothesis, we conclude that Claim 14 holds for B.

Proof of Lemma 4-5 Thus Claims 13–17 have been established for B prior to the writing of $\mathsf{initseq}(B)$ at state δ. It only remains to show that no reachable tuple is removed from B by the application of $\mathsf{initseq}$. This is argued by contradiction. Suppose a reachable tuple $\langle p, , m\rangle \in B$ is discarded by $\mathsf{initseq}$. This implies the existence of a "gap," i.e., for some distance ℓ, $\ell < m$, no tuple $\langle , , \ell\rangle \in B$ exists. All tuples contained in B have distances smaller or equal to tuples contained in e_q, by Claim 17. It follows that such a gap is the result of decreasing the distance of some tuple(s). This situation leads to the claim:

Claim 18 $(\forall \langle r, , j\rangle \in B : \; j < \ell \;\wedge\; r \neq z : \; \mathit{dist}_{\mathcal{T}}(r,p) = \infty)$

Claim 18 follows from Claim 16: on one hand, if $\mathit{dist}_{\mathcal{T}}(r,p) < (m-j)$ holds for any tuple $\langle r, , j\rangle \in B$, $j < \ell$, then the tuple $\langle p, , m\rangle \notin B$; on the other hand, if $\mathit{dist}_{\mathcal{T}}(r,p) \geq (m-j)$ holds for every tuple $\langle r, , j\rangle \in B$, $j < \ell$, then tuples at distances $m, (m-1), \ldots$ are by Claim 16 present in B and there is no gap at distance ℓ. As a consequence of Claim 18, there is some tuple $\langle p, , m\rangle \in B$ for which $\mathit{dist}_{\mathcal{T}}(q,p) = \infty$. Therefore $\langle p, , m\rangle \in B$ holds, because some neighboring processor's e register contained tuple $\langle p, , (m-1)\rangle$, which implies $\langle z, , \rangle \in B$. If $p = z$, then there exists some neighbor of z, call it s, so that $\mathit{dist}_{\mathcal{T}} = (q,s) = \mathit{dist}_{\mathcal{U}}(q,s)$, which by Claims 13 and 15 and the assumption that σ is legitimate for $\mathcal{T}$ contradicts the assumption of a gap. If $p \neq z$, then the tuple $\langle z, , \rangle$ has smaller distance than m, and by Claim 17 the existence of a gap is contradicted.

Proof of Lemma 4 □

Acknowledgement of support: Part of Shlomi Dolev's research was supported by TAMU Engineering Excellence funds, by NSF Presidential Young Investigator Award CCR-9396098, and by the Israeli Ministry of Science and Arts Grant 6756195.

Ted Herman's research was supported in part by the Netherlands Organization for Scientific Research (NWO) under contract NF 62-376 (NFI project ALADDIN: Algorithmic Aspects of Parallel and Distributed Systems).

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 358–370, New York, 1987. IEEE.

[AB97] Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, 1997. Society for Industrial and Applied Mathematics.

[AEYH92] E. Anagnostou, R. El-Yaniv, and V. Hadzilacos. Memory adaptive self-stabilizing protocols. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 203–220, Amsterdam, 1992. Elsevier.

[AG94a] Y. Afek and E. Gafni. Distributed algorithms for unidirectional networks. *SIAM Journal on Computing*, 23(6):1152–1178, 1994.

[AG94b] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[AGH90] B. Awerbuch, O. Goldreich, and A. Herzberg. A quantitative approach to dynamic networks. In *Proceedings of the 9th Annual*

ACM Symposium on Principles of Distributed Computing, pages 189–203, Quebec City, 1990. ACM.

[AGR92] Y. Afek, E. Gafni, and A. Rosen. The slide mechanism with applications in dynamic networks. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 35–46, New York, 1992. ACM.

[AK93] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Berlin, 1993. Springer-Verlag.

[AM92] B. Awerbuch and Y. Mansour. An efficient topology update protocol for dynamic networks. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 185–201, Berlin, 1992. Springer-Verlag.

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, New York, 1991. IEEE.

[CYH91] N. S. Chen, H. P. Yu, and S. T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[DH95] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems (preliminary version). Technical Report TR-95-02, The University of Iowa Department of Computer Science, 1995.

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

[DIM97] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self stabilizing message driven protocols. *SIAM Journal on Computing*, 26(1):273–290, 1997.

[Dol93] S. Dolev. Optimal time self stabilization in dynamic systems. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, pages 160–173, 1993.

[KP93] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:91–101, 1993.

[SG89] J. Spinelli and R. G. Gallager. Event driven topology broadcast without sequence numbers. *IEEE Transactions on Communication*, 37(5):468–474, 1989.