SuperStabilizing Protocols for Dynamic Distributed Systems

(Preliminary Version)

Shlomi Dolev*
Texas A&M University
shlomi@cs.tamu.edu

Ted Herman†
University of Iowa
herman@cs.uiowa.edu

January 1995

Abstract

Two aspects of reliability of distributed protocols are a protocol's ability to recover from transient faults and a protocol's ability to function in a dynamic environment. Approaches for both of these aspects have been separately developed, but have drawbacks when applied to an environment that has both transient faults and dynamic changes. This paper introduces definitions and methods for addressing both concerns in the design of systems.

A protocol is *superstabilizing* if it is (*i*) self-stabilizing, meaning that it is guaranteed to respond to an arbitrary transient fault by eventually satisfying and maintaining a *legitimacy* predicate, and (*ii*) it is 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 towards re-establishing legitimacy following a topology change.

Specific contributions of the paper include: superstabilizing protocols for coloring and spanning tree construction; metrics for evaluating superstabilization; 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.

*Part of this research was supported by TAMU Engineering Excellence funds and by NSF Presidential Young Investigator Award CCR-9396098.

†This 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).

1 Introduction

The most general technique to enable 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 since arbitrary transient faults can falsify any non-trivial safety property.

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 cope with such failures and recovery without global reinitialization. These protocols consider only global states that are reachable from a predefined initial state under a *restrictive sequence of failures*; under such an assumption, the protocols attempt to cope with failures with as few adjustments as possible. Thus, whereas self-stabilization research largely ignores the behaviour of protocols between the time of a transient fault and restoration to a legitimate state, dynamic protocols make guarantees about behaviour at all times (e.g. the period between a failure event and the completion of necessary adjustments).

1.1 Superstabilization

Superstabilizing protocols combine benefits of both self-stabilizing and dynamic protocols. We retain the idea of a legitimate state, but partitition illegitimate states into two classes, depending on whether or not they satisfy a *passage* predicate. Roughly speaking, a protocol is superstabilizing if it is (i) self-stabilizing, and (ii) when 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.

The passage predicate is defined with respect to a class of topology changes. 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 a topology change event. 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 lose the token but not falsify the passage predicate. Similarly, for the leader election task, the passage predicate could specify that at most one leader exists.

Superstabilizing protocols are evaluated in several ways. Of interest are the worst-case convergence time, i.e., the time required to estabilish 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 is the maximum amount of time[1] it takes for the protocol to reach a legitimate state.

Superstabilization time is 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.

Adjustment measure is 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

Many distributed protocols have been designed to cope with continuous dynamic changes (e.g. [AAG87, AGH90, AM92, AGR92]). These protocols make certain assumptions about the behavior of processors and links during failure and recovery; for instance, most of those works do not consider the possibility of processor crashes[2] and they assume that every corrupted message is identified and discarded. If failures are frequent, these restrictive assumptions can be too optimistic. In particular, when the protocol is an on-going protocol that does not stop running (e.g., distributed operating system, topology update, token-passing), even a single violation of the assumptions on the behavior of processors and links can cause the system to permanently be in an inconsistent state.

A number of researchers [DIM93, KP93, APV91] suggest a self-stabilizing approach to deal with dynamic systems. In these approaches, 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 deal with dynamic systems, the primary goal of self-stabilizing protocols is to recover from transient faults, and 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 — only convergence to a legitimate state is guaranteed.

Self-stabilization's treatment of a dynamic system differs from that of the dynamic protocols cited above in the way that topology changes are modelled. 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.[3] If a topology change can occur without any

[1] Measured by asynchronous time units called *rounds*, which are defined in the sequel.

[2] For example they do not tolerate the loss of the alternating bit value used by the alternating bit protocol [BSW69], nor the loss of the message that is been currently sent, see [DW93].

[3] Any such signal would be recorded in the state, but a (transient) faulty topology change could occur with no evidence of a signal.

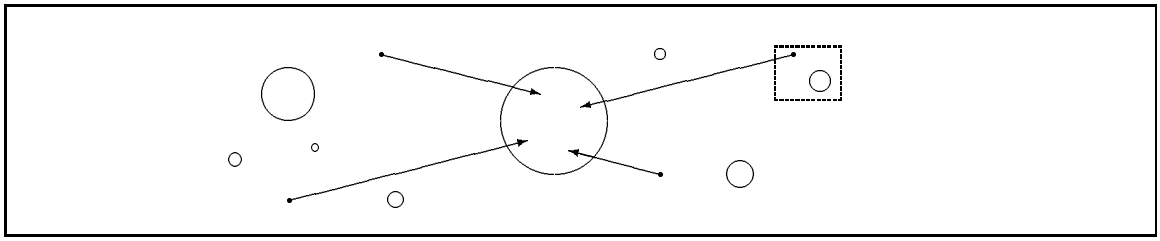

Figure 1: State Space and Convergence.

guarantee of a signal, it makes no sense to guarantee behavior of the protocol during the period following a topology change. Yet when the system is in a legitimate state and a fault *is* 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?

In addition to constraining a protocol's behaviour following a topology change to satisfy a safety property, we can also require that the adjustments processors make to converge are minimal. Consider the diagram shown in Figure 1. In this diagram, circles represent closed sets of legitimate states. Arrows represent convergence from an illegitimate state to a legitimate state. The length of an arrow is proportional to the scope of adjustment (e.g., number of processors that change state) due to convergence. The diagram illustrates the case of a self-stabilizing protocol that, upon detecting an illegitimate state, resets the global state to restore legitimacy. Notice, however, that one of the illegitimate states is near to a legitimate one, as highlighted by the dashed box. Instead of using a standard global reset, it would be better to adjust processor states so that convergence to a nearby legitimate state occurs.

The issue can also 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.

The time complexity of a self-stabilizing protocol is the worst-case measure of the time spent reaching a legitimate state from an arbitrary initial state. But is this measure appropriate for the view of self-stabilization for dynamic systems? Perhaps a better measure would be the worst-case of starting from an arbitrary legitimate state, considering a single topology change, and then measuring the time needed to again reach a legitimate state. This approach can be motivated by considering the probability of certain types of faults: while a transient fault is rare (but harmful), a dynamic change in the topology may be a frequent event. Note that the concern of time complexity is orthogonal to the concern of whether a protocol converges with minimal adjustments to its processor states following a

topology change.

1.3 Results and Comparison with Previous Work

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 [DIM93, KP93, APV91] do not use the fact that processor can detect that it is recovering following a crash (note that in [APV91] only link failures are considered); consequently there is no possibility of executing an "initialization" procedure during this recovery.

Recent work has shown how the basic model for self-stabilizing protocols, that consider transient-faults, can be extended to handle permanent-faults [AG92, GP93, DW93]. Such faulty behaviors as link crashes can be represented in the model by certain input variables; for instance, the neighborhood of a processor by an input variable containing a set of neighboring processor identifiers. A key observation 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 fall-back mechanism to deal with the change. In order to capture the possibility of reacting to a change we extend the definitions of [AG92, GP93] to include *interrupts* associated with such changes.

We use the fact that most dynamic changes are not entirely arbitrary, but are constrained, with fewer possibilities for change than in the general case of a transient fault. In particular we show *superstabilizing* protocols are able to respond to dynamic changes with few adjustments, in some cases only at a single processor. This situation corresponds to the intuitive notion that a distributed system should distributively adjust to topology change: if possible a change in one place of the system should not effect the output of the entire system.

Superstabilizing protocols can be directly constructed: we present examples in Sections 6 and 7. The general question of converting a self-stabilizing protocol to one that is superstabilizing is addressed by a method described in Section 9, which begins with a self-stabilizing protocol, then builds upon that protocol by adding new components that detect a dynamic change and then bring the system to a legitimate state with few adjustments. In case the dynamic change is too drastic (e.g. many simultaneous topology changes) it may be that our new components cannot cope with the change, in which case the underlying self-stabilizing protocol guarantees eventual convergence to some legitimate state.

The remainder of the paper illustrates, with protocols, how the handling of dynamic changes can be incorporated into protocol design. Following the introduction, we present in Sections 6 and 7 motivating examples. Section 2 formalizes the treatment of dynamic change. Section 9 describes a general method for converting a given self-stabilizing protocol into one that is optimized for dynamic change; this can be seen as a "dynamic optimizer" for self-stabilization. Finally, Section 11 contains concluding remarks.

2 Dynamic System

This section introduces notation and definitions of computation, dynamic change, stabilization and complexity measures. The general setting is a system in a dynamic environment. The state of the system has two components: one component consists of all the variables, program counters and communication data that can be altered by execution of system actions; the other component consists of input variables that represent the configuration of the system: these input variables cannot be changed by the system, but may be changed by the dynamic environment at any instant.

A system is represented by a graph where processors are nodes and links are (undirected) edges. An edge between two processors exists iff the two processors are *neighbours*; processors may only communicate if they are neighbours. 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; Section 13 sketches an implementation of the register model in terms of message-based constructions.

Associated with each processor are code, internal variables, program counters, and a shared register. A processor can write to its own shared register, but may only read shared registers belonging to neighbouring processors. The code of a processor is a sequential program; a program counter is associated with each processor. To simplify presentation, we make 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. 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. For many 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. For each processor p there is an input variable N_p, which is a list of processors q that are neighbours of p. Invariantly, neighbourhoods satisfy $p \notin N_p$ and $q \in N_p \Leftrightarrow p \in N_q$.

Local variables of processors 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 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 operation, for instance write$(e_p := 1)$ sets field image e_p and writes to

p's register.

The state of a processor p fully describes its internal state, its neighbourhood N_p, and the value contained in its register; 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 $\Theta = (\theta_1, \theta_2, \cdots)$ such that for $i = 1, 2, \cdots$ 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 infinite and contains infinitely many steps of each (non-crashed) processor.

A system *topology* is a specific system configuration of links and processors. Each processor can determine the current status of its neighbourhood from its local state (via N_p), so the system topology can be extracted 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 neighbourhoods and possibly removing or adding processors.

A topology change *event* is the removal or addition of a single link or processor, together with the execution of certain atomic steps specified in the sequel. Topology changes involving numerous links and processors can be modelled 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. In our model, a processor crash and a link crash are indistinguishable to a neighbour of the event: if p and q are neighbours and crash_p occurs, then we model this event by crash_{pq} with respect to reasoning about processor q. 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 iff p is incident on $\mathcal{E}$. Note that recovery of a processor together with links to its adjacent processors is treated as multiple events in our model: recov_p is one event, and each recov_{pq} for neighbouring q is a separate event; we further suppose that recov_p occurs prior to recov_{pq} in any processor (and neighbouring link) recovery sequence.

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 the program. Note that if $\mathcal{E}$ is incident on numerous processors, then all incident neighbourhoods change to reflect $\mathcal{E}$ and all processors execute the first interrupt step atomically with event $\mathcal{E}$. Thus the transition by $\mathcal{E}$ from $\mathcal{T}.\alpha$ to $\mathcal{T}.\beta$ changes more than neighbourhoods; states α and β also differ in the local states of processors incident on $\mathcal{E}$ due to execution of interrupt steps at these processors.

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 Stabilization

Researchers in the area of self-stabilization have proposed two sorts of definitions for the basic concept of legitimacy. The approach of [Dij74] defines legitimacy in terms of a predicate over the system state; the other approach [LL90] defines legitimacy in terms of behaviour. The paper [BGM93] shows that the behaviour approach, which defines legitimacy as a suffix property of computations, does not always have an equivalent expression in terms of predicates over system states. Although the behaviour approach may be more general, most stabilizing algorithms rely on some predicate over states (or snapshots of states) to initiate or control stabilization. We follow the approach of defining legitimacy in terms of a predicate on states. The remainder of this section defines legitimacy in our model and points out some advantages and disadvantages of our definition.

Each global state of a system can be classified as either legitimate or illegitimate: the predicate $\mathcal{L}$ holds iff the system is in a legitimate state. The notation $\sigma \vdash \mathcal{L}$ denotes that $\mathcal{L}$ holds at state σ. A protocol is *self-stabilizing* iff for any fair computation starting from an initial state α, $\alpha \vdash \mathcal{L}$, every state σ in that computation satisfies $\sigma \vdash \mathcal{L}$; and for every fair computation starting from any initial state α such that $\alpha \vdash \neg\mathcal{L}$, a state σ satisfying $\sigma \vdash \mathcal{L}$ is reached after a finite number of atomic steps.

Our model of processors and registers differs from the simpler state-reading model originally employed [Dij74] to define self-stabilization. In the state-reading model, there is no notion of a program counter; the protocol is a set of rules and in one atomic step a processor can read its own variables, the variables of its neighbours and write new values into its variables. Consequently, it is convenient in the state-reading model to describe predicate $\mathcal{L}$ as a relation over processor variables. In our model, such a predicate is more complicated since processor states consist of registers, local variables, and program counters; it is not so convenient to specify $\mathcal{L}$ completely as a relation over state-variables. Yet in nearly all cases, the essence of a legitimate state is captured by a predicate $\mathcal{P}$ over a limited subset of state-variables. For instance, $\mathcal{P}$ may hold if certain register fields form a tree in the network, or if at most one token exists in the system. The problem is that $\mathcal{P}$ may not itself be stable. It may be that a system state satisfies $\mathcal{P}$, but some local variables and program counters in that state are such that following one atomic step, registers are overwritten with the result that $\mathcal{P}$ is false. What is needed for a definition of $\mathcal{L}$ is a predicate that specifies the "reasonable" configurations of program counters, local variables and registers. Instead of explicitly defining $\mathcal{L}$ to cover all the details of program counters and local variables, we use the following technique. For any fair computation Φ and global state σ, $\sigma \in \Phi$, let $successor(\sigma)$ be the state following σ in Φ. Suppose $\mathcal{P}$ is the property of interest, e.g. $\mathcal{P}$ can be a predicate that holds if certain register fields form a tree. Then $\mathcal{L}$ is defined to be

the weakest solution in unknown predicate $\mathcal{X}$ of the equation

$$(\forall \Phi, \sigma : \sigma \in \Phi : \ (\sigma \vdash \mathcal{X} \Rightarrow \sigma \vdash \mathcal{P}) \ \wedge \ (\sigma \vdash \mathcal{X} \Rightarrow \mathit{successor}(\sigma) \vdash \mathcal{X}) \ )$$

For protocols in this paper, we use this technique to define $\mathcal{L}$ implicitly, in terms of some desired property $\mathcal{P}$, thereby not bothering to specify details over program counters and local variables. In order to prove that a protocol is self-stabilizing, we typically reason that a processor's local variables have sensible values after a single cycle in which neighbouring registers are read and the processor's own register is written using the local variables — since registers are written by only one processor, the local variables continue to accurately reflect the register contents after the first cycle. The characterization of program counters and local variables remains implicit in such a proof and spares the reader of details.

4 Superstabilization

One motivation for superstabilization is that a system should react gracefully to a topology change — preserving a passage predicate in the presence of the topology change. 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 procotol 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 until the protocol converges to a legitimate state.

Definition 4.1 A protocol P is *superstabilizing* with respect to Λ iff P is self-stabilizing and for every trajectory Φ beginning at a legitimate state and containing a single topology change events of type Λ, the passage predicate holds for every $\sigma \in \Phi$. □

Definition 4.2 A protocol P is *continuously superstabilizing* with respect to Λ iff P is self-stabilizing and for every trajectory Φ beginning at a legitimate state and containing only change events of type Λ, the passage predicate holds for every $\sigma \in \Phi$. □

Definition 4.2 is called *continuous superstabilization* because the environment is allowed to change the topology continuously, whereas Definition 4.1 addresses the case of a single topology change. The definition of continuous superstabilization is motivated by the approach of dynamic protocols, which are designed to handle asynchronous topology changes as they occur during system execution. This approach is necessary for dynamic protocols, since they have no secondary mechanism for recovery from errors. Definition 4.1 is motivated by the approach of self-stabilizing protocols, which can recover from topology change provided the environment is stable for a "long enough" period following the change. Although Definition 4.1 considers trajectories with a single change, we emphasize that the intention is to handle trajectories with multiple changes (each change is completely accomodated before the next change occurs). Our definition could be modified to state this explicitly, however we have chosen this simpler form in order to streamline proofs.

A particular passage predicate is of special interest — we use it in our general method for converting self-stabilizing protocols to superstabilizing protocols. To motivate this predicate, suppose the system is in a legitimate state and a Λ-topology change occurs. As a result, the system can be in an illegitimate state in such a way that a processor not incident on the topology change, i.e. at some distance from the change event, must eventually change its local state in order for the system to again reach legitimacy. However at the instant of a topology change, processors not incident on that topology change appear *locally* to have legitimate states. To formalize the notion of local legitimacy we propose a number of definitions.

Let D be a set of processors. With respect to any state σ such that D is some subset of the processors of $\mathcal{T}.\sigma$, define $\mathcal{T}.\sigma[D]$ to be the subgraph of $\mathcal{T}.\sigma$ induced by D. Let $\sigma[D]$ denote a vector of local states corresponding to the processors in D.

Definition 4.3 The vector of local states $\sigma[D]$ is *locally legitimate* iff there exists a state α such that $\mathcal{T}.\sigma[D]$ is a subgraph of $\mathcal{T}.\alpha$, $\sigma[D]$ is a subvector of α, and $\alpha \vdash \mathcal{L}$. □

Because adjustment to a topology change may require changing state information at some distance from the topology change, some type of coordination is necessary to effect the adjustment. During execution of the adjustment coordination, further topology changes could occur (or even transient errors). We are therefore interested in self-stabilizing coordination procedures, that is, procedures that are guaranteed to terminate coordination activity from any initial state. The following definition is a building-block for the synchronization of coordination following a topology change. It introduces the notion of a *filter predicate*, which is a locally evaluated predicate that intuitively represents activity of coordination following a topology change. The definition requires that filter predicates stabilize to *false* in any computation, meaning that all coordination activity eventually halts.

Definition 4.4 A predicate $\mathcal{I}_p$ is a filter predicate for protocol P iff $\mathcal{I}_p$ is a function mapping a local state of p to a boolean so that every fair computation has a suffix in which $(\forall p :: \neg\mathcal{I}_p)$ holds at every state in that suffix. □

The following definition specifies a particular form of passage predicate, one that insures local legitimacy during convergence. In the definition, the trajectory is left anonymous, since the definition could apply to either continuous or non-continuous superstabilization.

Definition 4.5 Passage predicate $\mathcal{Q}$ is *filter-based* iff there exists a filter predicate $\mathcal{I}$ such that for every state σ the following holds: $\sigma \vdash \mathcal{Q}$ iff for every set of processors D: if $\sigma \vdash \neg\mathcal{I}_p$ holds for all $p \in D$, then $\sigma[D]$ is locally legitimate. □

In words, for a filter-based passage predicate, any subgraph of the network in which filter predicates are false is locally legitimate. The usefulness of such a passage predicate depends

on a method to control the state of the filters by means of interrupts. The following remarks outline the use of our conventions for interrupts, filter predicates, and programs to achieve superstabilization. A change event $\mathcal{E}$ initiates an interrupt for all incident processors; atomically, the interrupt step of each incident processor sets some flag in a field variable and writes to its register so as to set $\mathcal{I}$ to *true* at that processor. Thus $\mathcal{I}_p$ is not only a function of the local state, but can be inspected by any neighbour. It remains only to guarantee that when a processor q reads a register indicating $\mathcal{I}_p = true$ for a neighbour p, then q immediately sets $\mathcal{I}_q$ to *true*. In this way, the condition on set D in the definition of superstabilization can be met. In essence, as the "news" of a topology change spreads in the network, processors are frozen before they can process this news and remain frozen until their states are adjusted to be made legitimate.

An advantage of a filter-based passage predicate is that $\mathcal{I}_p$ can be used locally to determine that a processor p is in a "vulnerable" state, that its local variables and registers are unreliable for the current topology. So long as $\mathcal{I}_p$ holds, a processor running a protocol P is potentially an unreliable provider of service. Our intent is that a client of P should take $\mathcal{I}_p$ into account when using P and wait until $\mathcal{I}_p$ is *false*, before relying again on P. Clients of P that take $\mathcal{I}_p$ into account enjoy a higher quality of service. The change from one topology to another is effectively atomic for such clients: the client switches from one legitimate state to another legitimate state without processing during an illegitimate state.

5 Complexity of Superstabilization

A primary contribution of 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 notion, we introduce an *adjustment measure*. To define an adjustment, we return to the notion of legitimacy and a property $\mathcal{P}$ 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. Now we consider a function $\mathcal{F} : \mathcal{O} \longrightarrow \mathcal{O}$. Function $\mathcal{F}$ maps states $\delta[\mathcal{O}]$ to states $\sigma[\mathcal{O}]$ satisfying $\sigma \vdash \mathcal{L}$, where δ and σ are any states such that σ can be obtained from δ 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 $\mathcal{E}$, 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 $\sigma[\mathcal{O}]$ and $\mathcal{F}(\sigma[\mathcal{O}])$, taken over all states σ 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.

To describe the time complexity of a protocol, the notion of a cycle is introduced. A *cycle* for a processor p is a minimal sequence of steps in a computation so that a complete iteration of the protocol at processor p completes, from first to last statement of the program

for p. All the programs of this paper are constructed so that a processor p's cycle consists of reading all of p's neighbour registers, 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 is finished at the first state at which every processor has completed at least one cycle; round $i+1$ terminates after each processor has executed one cycle following the termination of round i.

The order of magnitude of rounds, in terms of number of processors or network diameter, is the chief measure of time complexity. This permits us some freedom in the analysis of a protocol's cycles. For instance, we can generalize the definition of cycle to consist of some constant number of iterations of a processor's program. Typically, to analyse the round complexity of some protocol P, we consider a cycle to be a minimal sequence of steps so that the first through the last statement of program are executed in order. This means, for instance, that if P has a program with 30 statements, and execution begins at statement 15 (certainly possible in an arbitrary initial state), the first cycle would consist of execution of statements 15–30 followed by execution of statements 1–30, since this is the minimal sequence that guarantees that the first through last statement are executed in order (as opposed to 15–30 followed by 1–14).

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

6 Superstabilizing Colouring

This section exercises the definitions and notation developed in Section 2–5 for a simple allocation problem. A set of resources is to be allocated to processors so that no two neighbours share the same resource. The problem is challenging to the extent that the set of resources is limited. Our goal in this section is, however, not to investigate the most challenging instance of the general problem, but rather to illustrate aspects of superstabilization.

Let Δ be a parameter, intended as a bound the maximum number of neighbours for resource allocation. Let $\mathcal{C}$ be a totally ordered set of colours satisfying $|\mathcal{C}| \geq 1+\Delta$. Each processor p has a register field $colour_p$. The problem is to allocate colours to processors so that neighbouring processors have differing colours. We assume henceforth that each processor has at most Δ neighbours in any trajectory, which makes colour selection a simple matter. The property of interest $\mathcal{P}$ for legitimacy is

$$\sigma \vdash \mathcal{P} \;=\; \sigma \vdash (\forall\, p,q :\; q \in N_p :\; colour_p \neq colour_q) \;\wedge\; colour_p \in \mathcal{C} \;\wedge\; colour_q \in \mathcal{C}$$

A legitimate state for the colouring protocol is any state such that (i) property $\mathcal{P}$ is satisfied, and (ii) for each computation that starts in such a state, no processor changes colour in the computation.

The passage predicate $\mathcal{Q}$ for superstabilization is similar to $\mathcal{P}$, except that processors with $colour = -$ are not considered in conflicts:

$$\sigma \vdash \mathcal{Q} \;=\; \sigma \vdash (\forall\, p, q :\; q \in N_p :\; colour_p \neq colour_q \;\vee\; colour_p = - \;\vee\; colour_q = -)$$

The domain of a *colour* variable is thus extended to $\mathcal{C} \cup \{-\}$ to define the passage predicate.

Figure 2 presents a protocol for the allocation problem. Each processor has a local variables A and B used to collect the colours of its neighbours. A value $-$, satisfying $- \notin \mathcal{C}$ is introduced for the purposes of superstabilization. The function *choose* selects the minimum colour from the set S (and is undefined if S is empty).

The protocol of Figure 2 has two parts: one part is a self-stabilizing protocol, modified to deal with the $-$ element; the other part lists the interrupt that deals with topology change events. The self-stabilizing section perpetually scans for a colour conflict with the set of neighbouring processors having a larger identifier.[4] The interrupt statement writes to the register, conditionally changing the $colour_p$ field in case the topology change event was a restart of the processor or a link.

Lemma 6.1 Following one cycle of the colouring protocol at a processor p: $colour_p \in \mathcal{C}$ holds.

Proof: The lemma follows because the cycle of the self-stabilizing section includes steps S1–S7 in order, implying $|A| \leq \Delta$ and $|B| \leq \Delta$, from which we conclude that *choose* invocations in S8 and S9 deliver some colour from set $\mathcal{C}$. □

Lemma 6.2 The colouring protocol is self-stabilizing and converges in $O(n)$ rounds.

Proof: Let Φ be an arbitrary computation of the protocol beginning at state σ and let n be the number of processors in $\mathcal{T}.\sigma$. Lemmas 6.1 imply that after one round, no *colour* variable has or will obtain the value $-$ in the computation. We now show, by induction on the number of processors, that following round $2 + i$, $0 \leq i \leq n$, the i largest-identifier processors have permanent colour assignments such that no conflict with a neighbour of higher identity exists among these i processors. The basis for the induction is trivial since the empty set of processors satisfies the assertion. Now suppose the claim holds following round $2 + k$, $0 \leq k < n$. We examine the effect of round $3 + k$ with the respect to processor r, where r is the k^{th} largest processor identifier. In this round, processor r chooses some colour differing

[4] It is interesting to note that the more powerful state-reading model has a particularly simple protocol for colouring when the so-called central demon is assumed:

$$([]\, q :\; q \in N_p :\; colour_p = colour_q \;\;\rightarrow\;\; colour_p := choose(\mathcal{C} \setminus \{colour_r | r \in N_r\}))$$

is the rule for a process p; after at most n state transitions, where n is the number of processes, the protocol has stabilized. The notation $([]\, q :\; q \in N_p :\; X)$ is shorthand for specifying a copy of the rule X for each neighbour of p. This protocol fails under the distributed demon, i.e. when rules may simultaneously fire.

from any colour of a processor with a larger identity. The choice is deterministic, based on the colours of the larger identities. By hypothesis, these larger identity colour assignments are permanent, so following round $3+k$ and for all subsequent rounds, processor r's colour is fixed and differs from the colours of all neighbours of larger identity. The induction is completed. Thus after $2+n$ rounds, all processors have permanent colour assignments. It only remains to remark that no colour-conflict exists after $2+n$ rounds, since any such conflict would imply conflict between at least two processors, one having a larger identifier than the other. □

Lemma 6.3 The $O(n)$ bound of Lemma 6.2 is tight.

Proof: Consider a topology consisting of a chain of n processors, named $1, 2, \ldots, n$, with processors 1 and n being endpoints of the chain. Let $\Delta = 2$ and let $\mathcal{C}$ consist of three colours, $green < red < blue$. In the initial state, all processors have colour *red*. For the computation, we choose a scenario in which processors compute synchronously. In the first round, processor n chooses *green* because it is the minimum colour and all other processors also choose *green* because it is the minimum colour different from *red*. After the first round, $colour_n = green$ is stable for the remainder of the computation. In the second round, all processors other than n choose *red* since it is the smallest colour different from *green*. After the second round, $colour_{n-1} = red$ is stable for the remainder of the computation. The argument can be repeated to show that n rounds are required to reach a legitimate state. □

We conjecture that *any* self-stabilizing colouring protocol for dynamic systems with parameter Δ, where the number of processors may exceed Δ, has a worst-case convergence of at least $O(n)$ rounds (the network diameter is $O(n)$ in the worst case).

Lemma 6.4 The colouring protocol is superstabilizing with a superstabilizing time of $O(1)$ and adjustment measure $\mathcal{R} = 1$ for Λ being the class single topology change events.

Proof: Self-stabilization is proved in Lemma 6.2. Let Φ be a trajectory beginning at some state δ, $\delta \vdash \mathcal{L}$, with σ being the second state of Φ obtained from δ by a topology change event $\mathcal{E}$ (together with execution of the first interrupt step of all processors incident on $\mathcal{E}$), let Ψ be the suffix of Φ beginning with state δ, and let Ψ be a fair computation. The remainder of the proof consists of showing that the system reaches a new legitimate state without violating the passage predicate.

Every processor p with $colour_p \neq -$ in σ has an identical colour in δ. By the fact that σ is legitimate no two neighboring processors with $colour \neq -$ have identical colours. Next we show that only processors with $colour = -$ in σ change colours in Φ. For the cases crash_{pq}, or crash_p there is no processor with $colour = -$ in σ, so S8 is not executed. Moreover, the B set of colours in δ is a superset of B in any subsequent configuration in Φ. Thus, in any execution of S9 it holds that $colour_p \notin B$. Therefore, no processor change a color in Φ.

For the case of $\mathsf{recover}_p$ no conflict of colours may arise since there is no link connecting p to the rest of the system. The only left topology change is $\mathsf{recover}_{pq}$ for which a colour nonflict is eliminated by setting one colour to $-$. Then the single processor, p, with $colour_p = -$ in σ reads the neighbouring colours (in line S4) and assigns a colour that is not one of the neighbouring colours in line S8. Thus, no other processor change colour in Φ.

Both the $O(1)$ superstabilization time and the adjustment measure $\mathcal{R} = 1$ are implied directly from the above case analyzis. □

The colouring protocol illustrates *both* 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 transient fault or many topology changes rapidly occuring). Quantitative aspects can be seen by the $O(1)$ 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 colours $O(n)$ times during this convergence. Indeed if the superstabilizing components of the protocol are removed, namely S8 and the interrupt statement, then $O(n)$ rounds may be required for convergence following even a single topology change event starting from a legitimate state.

7 Superstabilizing Tree

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 to construct 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. Yet all the self-stabilizing spanning tree algorithms we know of require, in some cases (e.g. [DIM93], [AG90], [AK93]), recomputation of a tree when a link recovers, regardless of whether the network currently has a spanning tree or not. The reason is that a processor cannot locally "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. The protocol given in this section successfully "ignores" all dynamic changes that add links to an existing spanning tree or crash links that are not contained in the tree.

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.

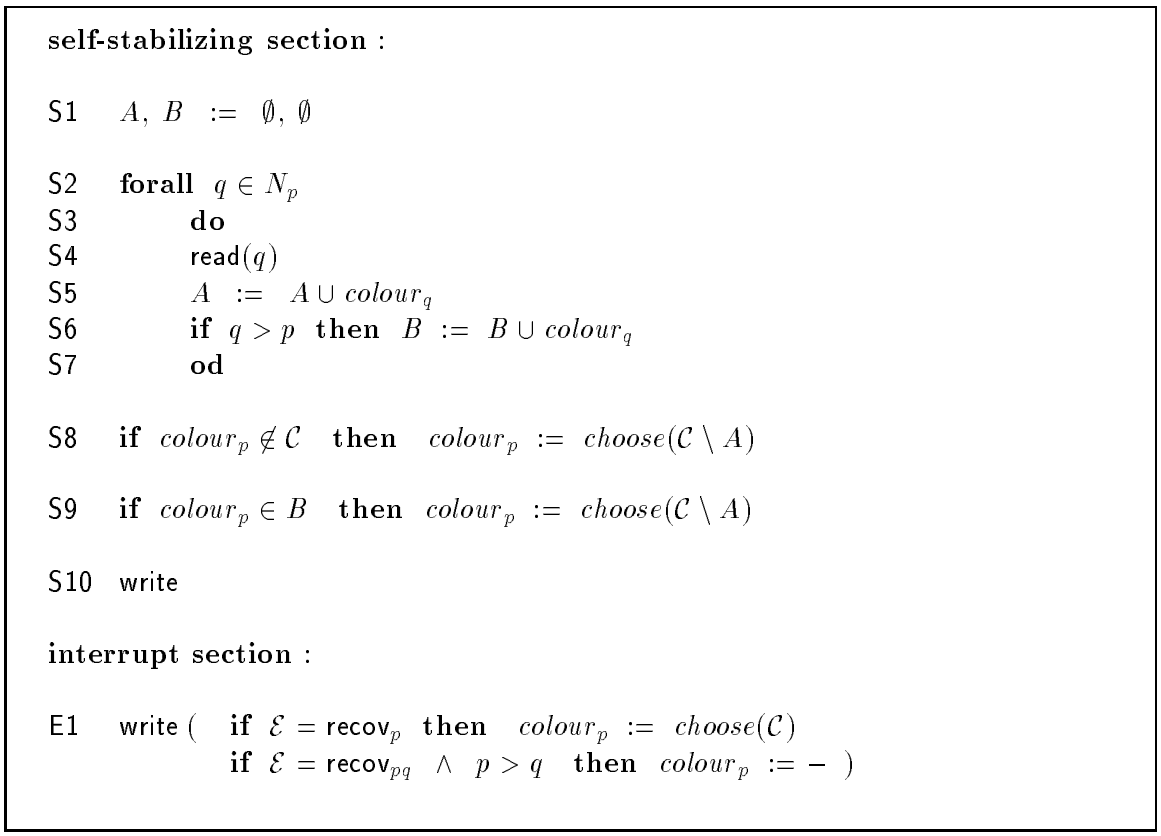

Figure 2: Superstabilizing Colouring Protocol for Processor p.

The basic idea of the protocol is the construction of a least-cost path tree to a processor r designated as the root of the tree. The key innovation 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. The register d_p contains a non-negative integer. The function w maps a pair of processor identifiers to an integer:

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

Figure 3 shows the code of the superstabilizing spanning tree protocol. The property $\mathcal{P}$ 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 (i) property $\mathcal{P}$ is satisfied, and (ii) for each computation that starts in such a state, no processor changes a t_p variable in the computation.

Lemma 7.1 The spanning tree protocol self-stabilizes in $O(n)$ rounds.

Proof: 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 satisfying: (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 as does $t_p \neq -$ for all p. Therefore, after the first round, T_r is non-empty, containing at least r. The remainder of the proof concerns rounds two and higher, and is organized into three claims.

> **Claim 1:** (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.

> **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 and also neighbouring T_r. Let p be such a processor, outside T_r and neighbour 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 neighbouring 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 lemma follows because there are at most n processors. □

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. The passage predicate $\mathcal{Q}$ for the superstabilization property is identical to $\mathcal{P}$.

Lemma 7.2 The spanning tree protocol is superstabilizing for the class Λ with superstabilization time $O(1)$ and adjustment measure $\mathcal{R} = 1$.

Proof: 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 non-tree 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 produce 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 produce identical results in any round following σ. Therefore $\sigma \vdash \mathcal{L}$. □

The tree protocol of this section illustrates quantitative and qualitative aspects of superstabilization. Since convergence occurs atomically with a change event from the class Λ, qualitative aspects of superstabilization are instantly satisfied — the system is always in a legitimate state! The quantitative aspects are due to the $O(1)$ superstabilization time and adjustment measure for changes in the class Λ.

The simple tree protocol of Figure 3 is not superstabilizing for events such as a tree link crash. Examination of this case reveals that the fragment of the tree that remains connected to the root following a link removal remains stable, which fulfills the goal of local adjustment in response to dynamic change. However to obtain a superstabilizing protocol, some machinery would be needed to control convergence following a tree link crash. Instead of developing such machinery for the specific task of tree construction, we tackle the general problem of superstabilization in subsequent sections.

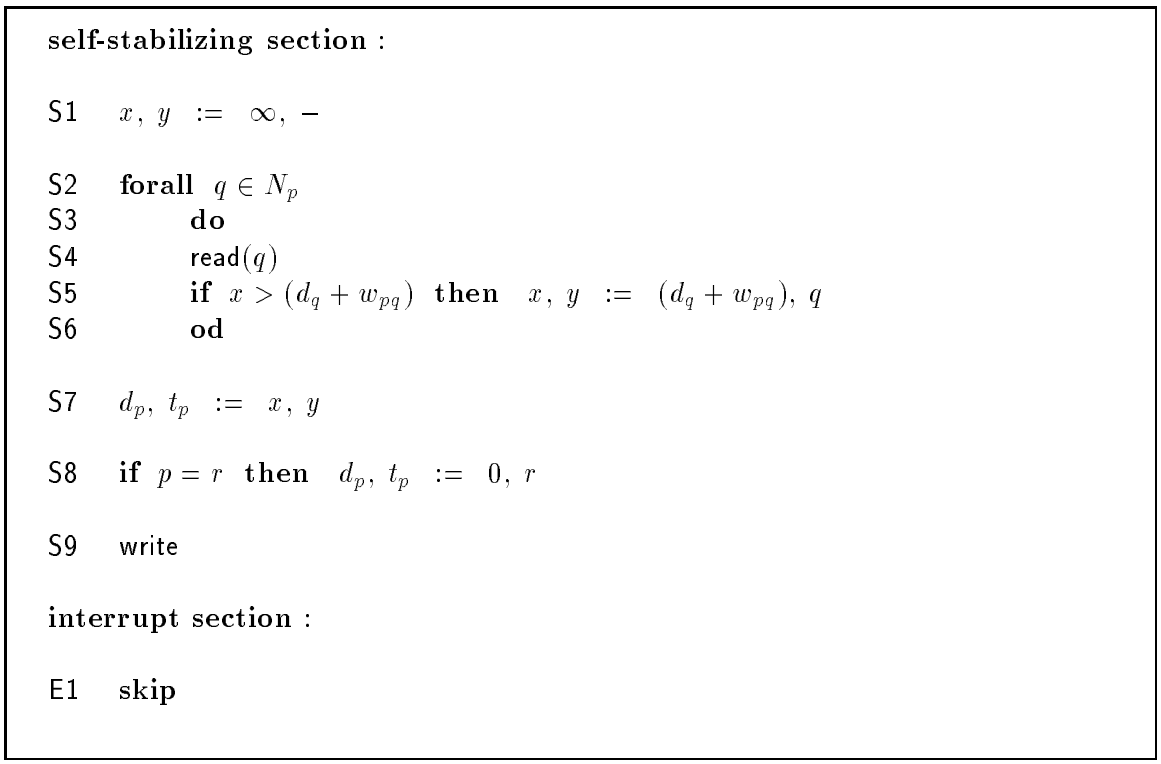

Figure 3: Superstabilizing Tree Protocol for Processor p.

8 Update Protocol

To simplify the presentation of our general methods for superstabilizing protocols, we employ a self-stabilizing update protocol. We view the update protocol as the simplest and clearest self-stabilizing protocol for large class of tasks including: leader-election, topology update and diameter estimation. 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 *topology update* when the field x_p contains all the local information about p's links and network characteristics. Many dynamic system 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, Do93]. To insure a desired deterministic property of the protocol, we assume that the neighbourhood of a processor N_p is represented as an ordered list.

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 $\mathsf{dist}_{\mathcal{T}}(p,q)$ be the minimum number of links contained in a path between processors p and q in topology $\mathcal{T}$; the third component of a tuple is intended to represent the dist-value for 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.

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 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 mindist is undefined.) Define $A \setminus\setminus \langle q, *, *\rangle$ to be the list of tuples obtained from A by removing every tuple whose first component is q. 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 $\mathsf{initseq}(A)$ by the following procedure: (1) sort the tuples of A in nondecreasing order of the third element of a tuple; (2) 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 $\mathsf{initseq}(A)$ is the set of tuples in this initial prefix.

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 4 presents the protocol.

Theorem 1 The update protocol of Figure 4 self-stabilizes in $O(d)$ rounds.
(Proof given in Appendix).

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 8.1 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 neighbours 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: The computation of B consists of at most nK tuples, since tuples with duplicate identifiers are not added to B by C8 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 C10 places at most nK tuples in e_p. The collection procedure to construct 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). □

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).

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 [AEH92]. The following lemma is an observation due to Gerard Tel.[5]

Lemma 8.2 The update protocol of Figure 4 is *memory-adaptive.*

Proof: Upon stabilization, the necessary size of the e-field is bounded by a function of the number of processors. □

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 x_p values observed at another processor q. 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

[5] Remark during presentation, December 1993.

that the protocol successfully stabilizes between changes to x_p, then a FIFO broadcast of x_p-values is obtained. In Section 9, we introduce an acknowledgement 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 acknowledgement 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 the next section to deal with the lack of dynamic monotonicity.

Theorem 2 The update protocol of Figure 4 enjoys static monotonicity.

The theorem can be proved by induction on a lexicographic measure composed of path length and the ordering of links by a processor's neighbourhood; 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.

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 3 The update protocol of Figure 4 enjoys impulse monotonicity.
(Proof appears in appendix, Section 12.)

Theorem 4 The update protocol of Figure 4 does not satisfy dynamic monotonicity.

A counter-example provides proof of this theorem and is presented in an appendix (Section 12). This 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.

9 General Superstabilization

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' that is both self-stabilizing and superstabilizing. This is done by modifying protocol P and superimposing a new component called the *superstabilizer*.

The superstabilizer makes use of function $\mathcal{F}$, described in Section 5, to determine a new legitimate state for protocol P following a topology change $\mathcal{E}$. It is the responsibility of

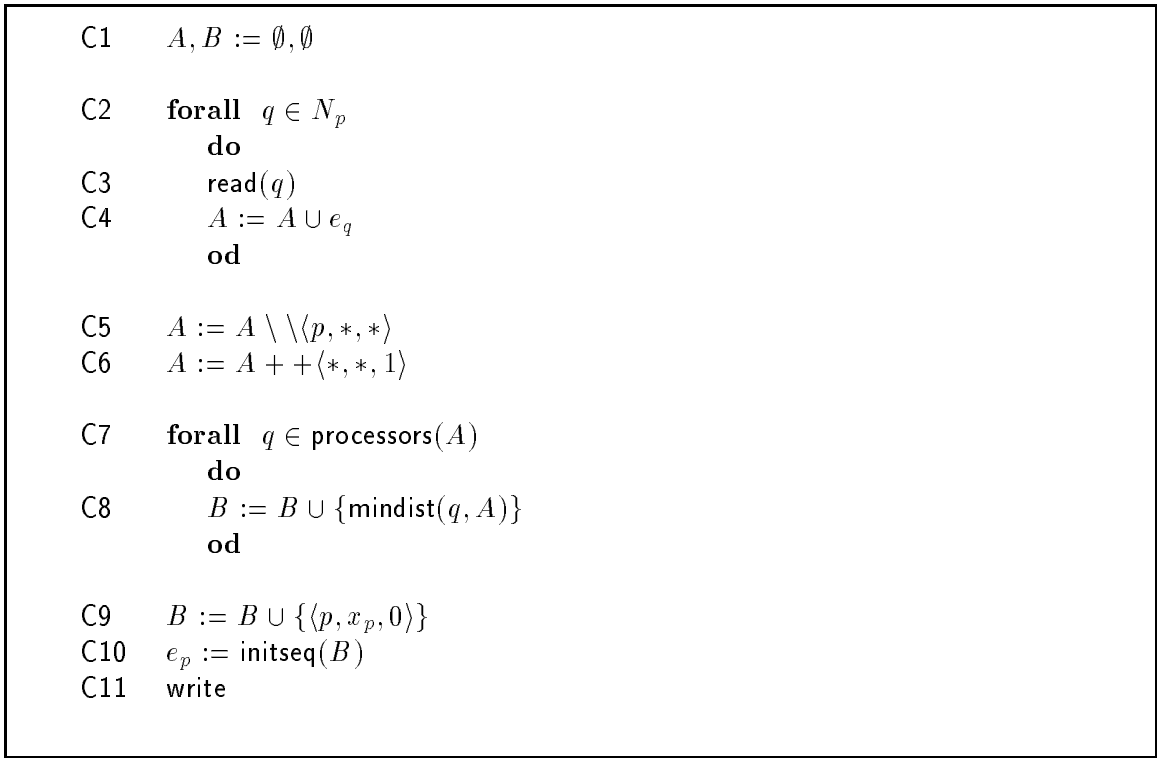

Figure 4: Update Protocol for Processor p.

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 procotol P from any stabilization effort. Thus the passage predicate for general superstabilization is a filter-based predicate.

The general method is a result primarily for the qualitative aspect of superstabilization; the superstabilization time is $O(d)$, which may not improve over the self-stabilization complexity of the original protocol P; however, the general method does make possible minimal adjustment following a topology change and provides the added benefit of a filter predicate that can be used by a consumer of P's service to delay use of the service during adjustment following dynamic change.

The superstabilizer consists of two components, a modified version of the update protocol and an interrupt statement. Atomic steps of P and the update protocol are then interleaved. 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 remains static. Our superstabilizer insures that, starting from any initial state, all *freeze* fields eventually become *false* in the absence of topology changes.

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.

Our 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 neighbouring processors; for example the code of Figure 4 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 convenience in describing the superstabilizer, we divide the x field into four subfields:

$$x_p \;=\; [\, a_p \; h_p \; t_p \; u_p \,]$$

We then extend the programming notation to allow any processor to refer to subfields of any other processor. Thus processor p can refer to a_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.

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:

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 $(\forall p :: a_p = 2)$ holds, and thereafter terminates in order to resume execution of Phase 0.

To detect progress of phases, we employ an acknowledgment subfield h_p. This subfield is a vector of ternary values whose elements are images known to p of other processor a-subfields: 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.

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. We also define the subfield u_p to contain a global *snap* value, i.e. $u_p[r]$ contains a snap_r value. We denote by s_p the collection of all t_r images obtained from x_p subfields.

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

The interrupt statement for the superstabilizer is given in Figure 5. In response to a topology change $\mathcal{E}$ incident on processor p, the program counter of the protocol is reset to S1, the neighbourhood 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*.

The remaining component of the superstabilizer consists of the combination of Figures 4 and 6, i.e. a modified update protocol. Statements U1–U8 should be inserted between statements C6 and C7 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. Some motivation for statements U1–U8 is given by:

U1 represents the election of a single coordinator from all the processors incident on the topology change event $\mathcal{E}$. Although all incident processors execute S1 and set $a_p = 1$, all but one of processor per connected component will revert back to $a_p = 0$ upon detection of a competing coordinator with a higher identity.

U2 is the transition from Phase 1 to Phase 2. This occurs when p is coordinating Phase 1 and detects that every other processor has either acknowledged Phase 1 or appears to have already acknowledged Phase 2 (because of an illegitimate initial state). At statement U2, a new global state is computed using adjustment function $\mathcal{F}$, based on the collected snapshots of all acknowledging processors.

U3 is the transition from Phase 2 back to Phase 0. This occurs when p is coordinating Phase 2 and detects that every other processor has either acknowledged Phase 2 or appears to have already acknowledged Phase 0 (because of an illegitimate initial state).

U5 acknowledges Phase 0.

U6 acknowledges Phase 1, but only if it appears that the last known phase for a coordinating processor was Phase 0. This will make the transition from Phase 1 to Phase 2 monotonic, since processors will not switch allegiance back to Phase 1 after seeing Phase 2.

U7 acknowledges Phase 2, but only if it appears that the last known phase for a coordinating processor was Phase 1. This will make the transition from Phase 2 to Phase 0 monotonic, since processors will not switch allegiance back to Phase 2 after seeing Phase 0. The step U7 also adopts a new local state as provided by the phase coordinator.

U8 insures that processors set *freeze* bits so long as there is phase activity by some coordinator.

Note that processors react to superstabilization phases as soon as they are made available via the update protocol; in particular, as soon as a (possibly distant) topology change is visible, the code of Figure 6 freezes protocol P and records a snapshot.

The combination of the superstabilizer and modified protocol P results in a superstabilizing protocol P'. A legitimate state for P' is any state in which: (1) the variables, fields and program counter with respect to P satisfy $\mathcal{L}$; (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*, $(\forall p :: a_p = 0)$; and (4) each computation that starts in such a state preserves (1)–(3).

Lemma 9.1 The predicate $f_p \equiv freeze_p$ is a filter predicate for the superstabilizer; and the superstabilizer converges in $O(d)$ rounds to $(\forall p :: \neg f_p)$.

Proof: First we show that the protocol stabilizes to $a_p = 0$ for every p in any computation; after such stabilization, it follows by self-stabilization of the update protocol that in $O(d)$ rounds, every image of a_p is also accurate. Then in one additional round statement U8 is executed at every processor, which implies stabilization to $(\forall p :: \neg f_p)$.

To show that the protocol stabilizes to $(\forall p :: a_p = 0)$, consider an arbitrary computation Φ. Observe that no statement of the code in Figure 6 can assign to a_p in the case that $a_p = 0$ is a precondition; in other words, $a_p = 0$ is locally stable. Therefore it suffices to show that eventually $a_p = 0$ is obtained for every p.

Suppose, heading for contradiction, that $a_p \neq 0$ holds for $p \in D$ throughout Φ, where D is some non-empty set of processors. Let Ω be a suffix of Φ in which no processor assigns $a_p := 0$. Thus statements U1 and U3 are not executed by any processor in Ω. After $O(d)$ rounds of Ω, all a_p images accurately and permanently allow any processor p to test $a_r = 0$ for any known r; let Ψ be the suffix of Ω with this property. Observe that $|D| = 1$ in Ω if the network's topology is connected. This follows because after $O(d)$ rounds of Ω, if $a_p \neq 0$ and $a_r \neq 0$ for $p \neq r$, then statements U6 and U7 executed at all processors insure $(\forall s :: h_s[p] \neq 0)$ and $(\forall s :: h_s[r] \neq 0)$; following another $O(d)$ rounds, the images of all h-fields are broadcast to p and r, which implies statement U1 is executed for one of the two processors, leading to a contradiction. Hence there is at most one processor p satisfying $a_p = 1$ in any connected component in computation Ω. If $a_p = 1$ then eventually the image of a_p is broadcast by the update protocol and via the acknowledgement of U6, the predicate $(\forall q :: h_q[p] \neq 0)$ holds and p can detect this using images of the h_q fields. That is, either a processor q will acknowledge $a_p = 1$ or permanently retain $h_q[p] = 2$. In either case, eventually U2 is executed for p. And if $a_p = 2$ then by a similar argument, U3 is eventually enabled, but this contradicts the definition of D.

That f_p is a filter predicate is shown by the above contradiction. It remains to show that convergence to $(\forall p :: \neg f_p)$ occurs in $O(d)$ rounds of any computation. In the case where a processor p is the only processor satisfying $a_p \neq 0$ within a connected component, the same update/acknowledge arguments given above show that in a constant number of broadcasts via the update mechanism, the protocol stabilizes to $a_p = 0$, which implies $O(d)$ convergence. In the case of numerous, competing processors satisfying $a_p \neq 0$, we appeal to arguments about statement U1 to conclude that a winner among the competing processors is obtained in $O(d)$ rounds. □

Lemma 9.2 The general method self-stabilizes in $O(d + K)$ rounds, where $O(K)$ is the worst-case stabilization time of P.

Proof: After $O(d)$ rounds, by Lemma 9.1, all *freeze* bits are permanently *false* in any computation. Thereafter the superstabilizer component does not hinder progress of protocol P, which reaches legitimacy by assumption in $O(K)$ rounds. □

Lemma 9.3 The protocol of Figures 5 and 6 is superstabilizing with superstabilization time $O(d)$.

Proof: Lemma 9.2 shows that the protocol is self-stabilizing; it remains to show for any trajectory beginning from a legitimate state followed by a single topology change, that in

every state of the computation any set of processors with *false* filter predicates is locally legitimate. The proof has the following structure. The computation following a topology change can be divided into three segments. In the first segment, the set of all processors with $f_p = \mathit{false}$ is locally legitimate with reference to the initial topology of the trajectory. In the second segment, $(\forall p :: f_p)$ holds. In the third segment, the set of all processors with $f_p = \mathit{false}$ is locally legitimate with reference to new topology.

Let Φ be a trajectory with initial state $\delta \vdash \mathcal{L}$ and initial transition due to some topology change event $\mathcal{E}$ resulting in a second state σ. Let Ψ be the suffix of Φ beginning at state σ (Ψ is a fair computation). Let H be the set of processors for which there exists a path in $\mathcal{T}.\delta$ to some processor incident on $\mathcal{E}$. All our reasoning about local states is confined to processors in H since if the network is partitioned throughout Φ, processors in components not related to $\mathcal{E}$ trivially remain in legitimate states with *false* filter predicates. Let $H[r]$ for any $r \in H$ denote the maximum subset $G \subseteq H$ such that $r \in G$ and the processors of G are connected in $\mathcal{T}.\sigma$. In case there are $r, s \in H$ so that $H[r] \neq H[s]$, then we may reason about the superstabilization of the components corresponding to $H[r]$ and $H[s]$ separately for the proof of superstabilization, since these components are not connected in $\mathcal{T}.\sigma$. For any $r \in H$, let $\mathit{leader}.H[r]$ be the processor v with maximum identity such that $v \in H[r]$ and v is incident $\mathcal{E}$.

The remainder of the proof is organized into six claims. To streamline the proof, we consider an arbitrary set of processors $H[r]$ and computations of these processors. Let v be $\mathit{leader}.H[r]$. The following predicates are defined for the claims:

$$\begin{aligned} Ph_1 &\equiv a_v = 1 \ \wedge \ (\forall q : q \neq v : a_q = 0) \ \wedge \ (\forall q :: h_q[v] = 1) \\ Ph_2 &\equiv a_v = 2 \ \wedge \ (\forall q : q \neq v : a_q = 0) \ \wedge \ (\forall q :: h_q[v] = 2) \end{aligned}$$

Claim 1: The following temporal[6] property is claimed for Ψ: for any processor p, $a_p = 1$ holds continuously in Ψ until processor p either observes Ph_1 or observes $(a_q = 1 \ \wedge \ (\forall r :: h_r[q] = 1))$ for some $q > p$ (note that just because p observes a condition from its images of other processor's a and h fields does not imply that the condition holds over the actual fields). The claim is a consequence of the conditions on statements U1 and U2.

A corollary of Claim 1 is that $a_v = 1$ holds until v observes Ph_1; moreover, when v observes Ph_1, it follows from U6 and U8 that, for every p, f_p holds at some previous state in Ψ. This observation is strengthened by the following claim.

Claim 2: For any processor p, f_p holds continuously in Ψ until processor v observes Ph_1. Statement U8 sets freeze_p in the presence of some image $a_q = 1$. Therefore we strengthen the claim to: processor p observes $(\exists q :: a_q = 1)$ holds continuously in Ψ until processor v observes Ph_1. Consider that a processor p observes some $a_q = 1$ for some q. By impulse monotonicity of the update protocol, the image $a_q = 1$ is stable so long as processor q does

[6] A definition of the **until** operator can be found in [CM88].

not change a_q. The only statements capable of changing a_q from the value 1 are statements U1 and U2. Statement U2 may change a_q only if every processor in $H[q]$ has acknowledged $a_q = 1$; if $q = v$ and statement U2 changes a_v, then v observes Ph_1, and the claim holds; if $q \neq v$ and statement U2 changes a_v, we have a contradiction since U1 is not executed in the same cycle and q is not the maximum identifier having $a_q = 1$. Therefore consider statement U1. If a_q is changed by U1, some other processor r with a larger identifier appears (at q) to satisfy $a_r = 1$ and $(\forall s :: h_s[r] = 1)$ holds. Thus a processor q cannot change a_q from 1 to 0 until first detecting that some other processor r of larger identifier has $a_r = 1$ and every processor has observed $a_r = 1$ at some point in computation Ψ. By induction, the predicate $(\exists s :: a_s = 1)$ as determined by images at every processor in $H[v]$ is stable at least until a_v is changed from 1 to 0.

Claim 3: In $O(d)$ rounds, a state satisfying Ph_1 is obtained; moreover, when v observes Ph_1, then Ph_1 holds and s_v contains a valid snapshot of the processors of $H[r]$. Claim 2 shows that f_p is stable for every $p \in H[v]$ at least until Ph_1 is observed by v (and Claim 1's corollary shows that f_p holds at every p). Consider any computation Θ beginning at state σ so that Ph_1 is not observed by v at any state. Induction over distance from v is now used to prove

$$(\forall q :: f_q \ \wedge \ h_q[v] = 1) \tag{1}$$

i.e. in round $k + 1$ all processors q at distance k from v satisfy the term of (1). The basis of the induction is distance zero: in state σ, from the interrupt statement, f_v holds, and after one round $h_v[v] = 1$ holds, by the assumpion of a legitimate state for δ, meaning $\delta \vdash h_v[v] = 0$. For the induction, suppose the term of (1) holds for all processors at distance k from v following round $k + 1$. In round $k + 2$, any processor q at distance $k + 1$ from v will read a tuple $\langle v, , k \rangle$ from a neighbour and update its own tuple for fields corresponding to v, therefore setting f_q and $h_q[v] = 1$.

To complete proof of the Claim 3, we observe that $a_v = 1$ in state σ and continues to hold at least until processor v observes $(\forall q :: h_q[v] = 1)$. However, for any competitor of v, namely a processor r so that $a_r = 1$ in state σ, the same cycle wherein $h_r[v] := 1$ occurs also sets $a_r := 0$ at statement U1. Statement U1 makes no assignment for processor v by assumption of v's maximum identifier. Finally, we observe that within computation Θ the predicate $h_q[v] = 1$ is stable since $a_v = 1$ is stable at least until (1) holds and the image $a_v = 1$ at any processor is stable by impulse monotonicity of the update protocol. Statement U8 insures that f_p is set and a snapshot is taken.

Claim 4: Let Ψ' be the segment of Ψ identified by Claim 2, that is, Ψ' begins at state σ and ends at some state where v observes Ph_1. The state α following the final state of Ψ' is obtained by the execution of U2, which sets $a_v := 2$ and computes a new state for the processors in $H[v]$. Let Ω be the segment of Ψ beginning at state α. Claim: for any processor p, f_p holds continuously in Ω until processor v observes Ph_2; also $h_p[v] = 2$ holds

continuously in Ω until v observes Ph_2. To show this claim, we consider any p and show that $h_p[v] = 1$ holds until $h_p[v] = 2$, and $h_p[v] = 2$ holds at least until v observes Ph_2. At the initial state of Ω, $h_p[v] = 1$ holds for any $p \neq v$ by construction. Now consider some processor that changes $h_p[v]$ from 1 to 2 in Ω. This can only be due to execution of U7; although the update protocol is not monotonic in the broadcasted change of a_v from 1 to 2, statement U6 does not assign 1 to $h_p[v]$ for any processor that has already executed U7 in Ω. Thus the change to $h_p[v]$ is monotonic within Ω. Also, although images of a_v may switch between 1 and 2 during the course of Ω, the predicate $a_v \neq 0$ remains stable, hence U8 cannot assign *false* to the *freeze* variable.

Claim 5: Within $O(d)$ rounds of computation Ω, a state satisfying Ph_2 is obtained; moreover, when v observes Ph_2, then Ph_2 holds and every processor in $H[v]$ has adopted the portion of the state computed by v via U2 at the state prior to Ω. Claim 4 shows that f_p is stable for every $p \in H[v]$ at least until Ph_2 is observed by v. Claim 4 also shows that $h_p[v] = 2$ is stable until v observes Ph_2. Since initially, $h_p[v] = 1$ holds at every p, the observation of $(\forall p :: h_p[v] = 2)$ implies each processor p has acknowledged the second phase and adopted a new state of protocol P via statement U7. The $O(d)$ round complexity can be shown by induction, similar to the argument given for Claim 3.

Claim 6: Let Σ be the segment of Ψ identified by Claim 5, beginning after execution of U3 at processor v. The claim is that for any processor p, $h_p[v] = 2$ holds continuously in Ω until $h_p[v] = 0$, that $h_p[v] = 0$ is stable, and $h_p[v] = 0$ is detected by p within $O(d)$ rounds. First observe that a corollary of Claim 5 is that no image of a_v satisfies $a_v = 0$ at the initial state of Σ. Now the logic of statements U5 and U7 prove the claim. After a processor p assigns $h_p[v] = 0$ via U5, subsequent execution of U7 is inhibited. Thus the switch from $h_p[v] = 2$ to $h_p[v] = 0$ is monotonic. The $O(d)$ round complexity can be shown by induction on distance from v.

Claims 1–6 together divide a computation following a topology change into segments. Once a processor p satisfies f_p, then f_p holds until p has received and assigned a new state for P; subsequently f_p continues to hold until $\neg f_p$ holds (implied by Claim 6), and then $\neg f_p$ is stable. The time for this computation is $O(d)$ rounds.

To complete the proof of superstabilization, we consider subsets $D = \{ p \mid p \in H[v] \wedge \neg f_p \}$ and verify that D is locally legitimate. For the segment Θ, each processor p for which $\neg f_p$ holds may execute protocol P. We show that local legitimacy holds for D by structural induction. The basis of the induction is the initial state σ, where all processors incident on $\mathcal{E}$ have P-states that are legitimate for $\mathcal{T}.\delta$. For the induction, we consider execution of protocol P within Θ. With regard to protocol P, a processor p may communicate with neighbours. Such communications fall into two categories: if p executes a **read** of a neighbour processor q for which $\neg f_q$ holds, then $\neg f_q$ holds continuously from the initial state of Θ up to that step; and if f_q holds, then q's state and registers with respect to protocol P are those

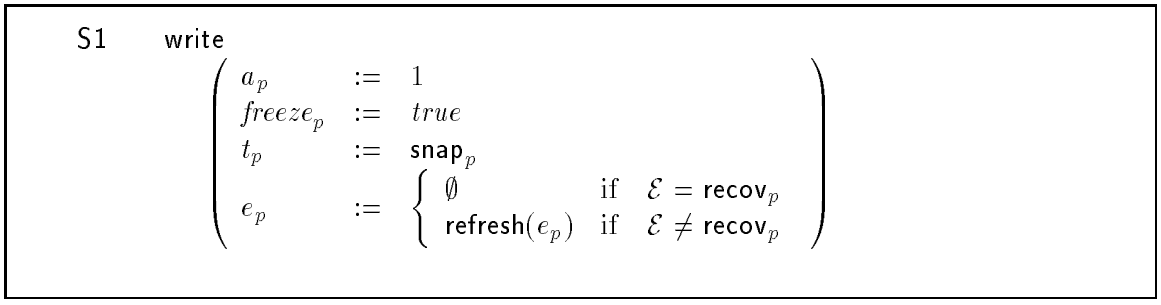

Figure 5: Superstabilizer: Processor p Interrupt Section.

of $\mathcal{T}.\delta$ by the inductive hypothesis. This argument discharges the proof of superstabilization for segment Θ; segment Ω need not be examined, since $(\forall p :: \neg f_p)$ holds at all states. It remains to examine D with respect to segment Σ. This is a simple matter since at the initial state of Σ, each processor p has a local state legitimate to $\mathcal{T}.\sigma$, and the setting of $f_p := \mathit{false}$ is implied monotonic by Claim 6. □

10 Continuous General Superstabilization

This section introduces a general method for achieving *continuous* superstabilization with respect to the class Λ of single topology changes (link or processor crash or recovery). The method described in this section is a generalization of the technique used in Section 9. The main difference is lies in the synchronization of multiple topology changes, which is achieved through the use of *incarnation numbers* associated with topology changes.

The general method for superstabilization described in Section 9 relies on the impulse monotonicity of the update protocol to coordinate the transition from a legitimate state in one topology to a legitimate state in another topology. In the presence of multiple topology changes between computations of a trajectory, or if a topology change should occur following a computation that has not yet reached a legitimate state, then impulse monotonicity is not strong enough to guarantee orderly coordination of the snapshot and reset mechanism. In order to enforce monotonicity of the update protocol, we introduce incarnation numbers associated with each topology change (and use processor identifiers to break ties).

The general method for continuous superstabilization is structured like the method of Section 9: it can be seen as a compiler that takes self-stabilizing protocol P and outputs a new protocol P' that is both self-stabilizing and continuously superstabilizing. The superstabilizer consists of two components, a modified version of the update protocol and an interrupt statement. For convenience in describing the superstabilizer, we divide the x field into six subfields:

$$x_p \;=\; [\, a_p \;\; h_p \;\; b_p \;\; g_p \;\; t_p \;\; u_p \,]$$

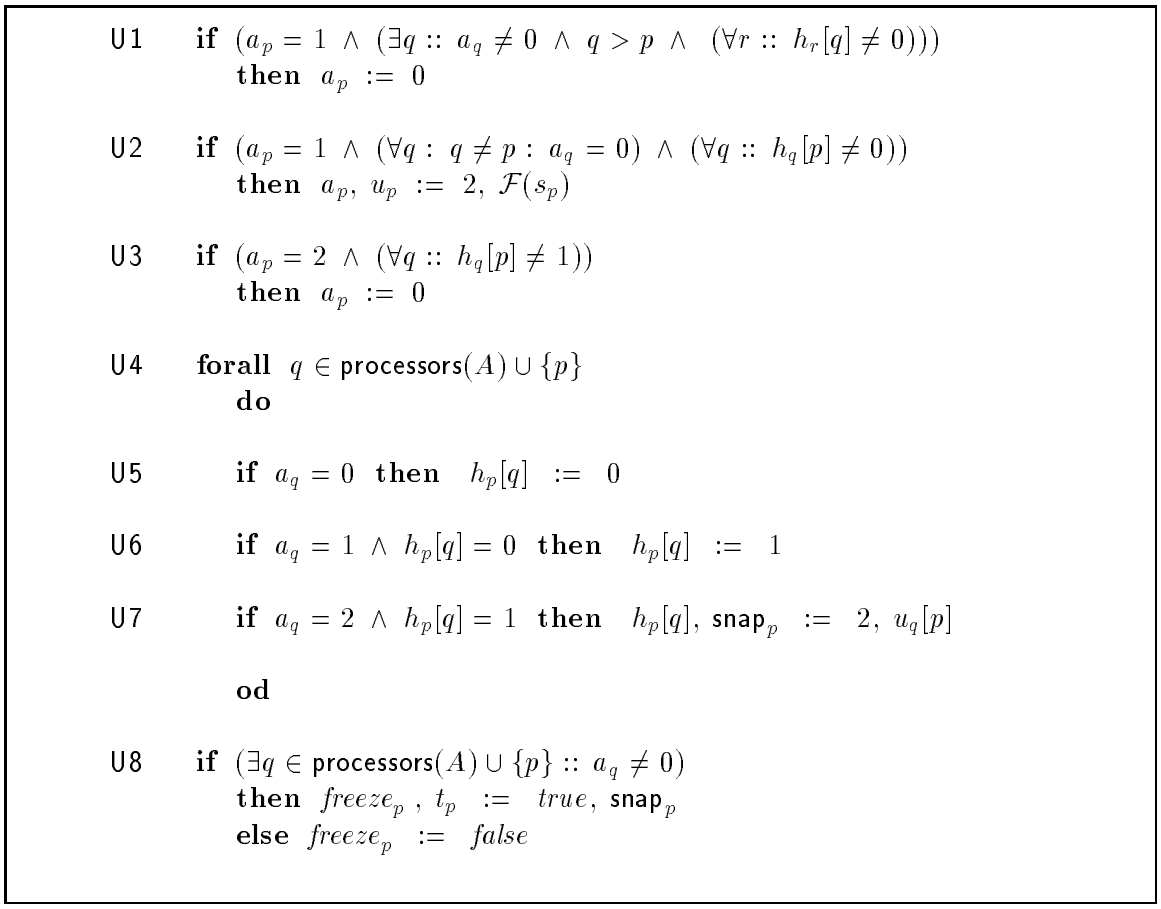

Figure 6: Superstabilizer: Update Extension for p.

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, as described in Section 9. Other fields are also as described in Section 9, with the exception of b_p and g_p fields, which are used for the incarnation numbers. The field b_p is an unbounded integer, used as a timestamp to synchronize concurrent topology changes. Intuitively, each topology change causes incident interrupt statements to initiate Phase 1 with an *incarnation number* (the timestamp b_p) that is greater than any previously known incarnation number. Processors under the control of multiple phase coordinators, possibly due to concurrent topology changes, should follow only the snapshots and resets from the coordinator having the largest incarnation number (breaking ties by processors' identifiers). The bookkeeping to insure that the most recent topology change has the largest incarnation number requires a recording technique similar to that used for phase coordination: g_p is a vector of integers whose elements record the largest b_q values observed by processor p.

The interrupt statement for the superstabilizer is given in Figure 7. In response to a topology change $\mathcal{E}$ incident on processor p, the program counter of the protocol is reset to S1, the neighbourhood 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*. Note that the interrupt step increments b_p to obtain a new incarnation number.

The remaining component of the superstabilizer consists of the combination of Figures 4 and 8, i.e. a modified update protocol. Statements U1–U12 should be inserted between statements C6 and C7 to obtain the complete protocol. To simplify presentation, the notation maxb_p is introduced:

$$\mathsf{maxb}_p \;\equiv\; \max(\,\{b_p\} \;\cup\; \{\,b_q \mid q \in \mathsf{processors}(e_p)\}\;\cup\; \{\,g_p[q] \mid q \in \mathsf{processors}(e_p)\}\,)$$

Thus maxb_p represents the maximum incarnation number known at processor p.

The code of Figure 8 essentially consists of two parts: statements U1–U4 keep the incarnation numbers of various processor consistent; statements U5–U12 control the phases of superstabilization. In particular:

U1 insures that an idle processor copies the largest known incarnation number.

U2 is a kind of election: if a processor is coordinating the phases of superstabilization, but encounters another competing processor with a larger incarnation, then it abandons superstabilization (yielding to the larger incarnation).

U3 covers a case not normally possible starting from a legitimate state: a processor coordinating the phases of stabilization encounters an idle processor with a larger incarnation number; in this case, the coordinating processor restarts the phases using a new, higher incarnation number. Statement U3 is crucial to the self-stabilizing property of the superstabilizer — coordinating processors are guaranteed to eventually become idle (in the absence of topology changes).

U4 essentially repeats the logic of U2, but the election occurs because two competing processors have the same incarnation number: the one with the larger identifier wins.

U5 is the transition from Phase 1 to Phase 2 by the coordinating processor; this occurs when every processor has acknowledged Phase 1 and transmitted a snapshot. Here a new global state is computed and broadcast via u_p.

U6 is the transition from Phase 2 back to Phase 0. This transition also produces a new, higher incarnation number. The incremented incarnation number helps in insuring monotonicity in the broadcast of register fields via the update protocol.

U8 shows how p keeps track of the incarnation number of every processor q. Thus $g_p[q]$ represents the largest incarnation number recorded by p for processor q. If b_q is lower than expected at processor p, then p will refrain from acknowledging Phase 0 by q.

U9 is where p acknowledges that q is in Phase 0, but only if q has a reasonable (not too low) incarnation number. This is the trick to make things monotonic, as far as phases go — recall that the interrupt atomically increments b_p and sets Phase 1, so no process will go back to Phase 0 once it records a larger incarnation number in $g_p[q]$.

U10 is the acknowledgement of Phase 1; this is only permitted when the coordinating processor appears to be in Phase 0 (to insure monotonicity of phase transition).

U11 is the acknowledgement of Phase 2 and adopting a new local state set by the coordinating processor of Phase 2. Note that Phases 1 and 2 are monotonic — once a processor acknowledges Phase 2, it will ignore any news of Phase 0 or Phase 1 from that processor at the current incarnation number.

U12 insures that any phase activity freezes a processor.

The combination of the superstabilizer and modified protocol P results in a superstabilizing protocol P'. A legitimate state for P' in a topology $\mathcal{T}$ is any state in which: (1) the variables, fields and program counters with respect to P satisfy $\mathcal{L}_{\mathcal{T}}$; (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*, $(\forall p :: a_p = 0)$; and (4) each computation that starts in such a state preserves (1)–(3).

Lemma 10.1 The predicate $f_p \equiv freeze_p$ is a filter predicate for the protocol; and the protocol converges in $O(d)$ rounds to $(\forall p :: \neg f_p)$.

Proof: We show the stronger property that the protocol stabilizes to $a_p = 0$ for every p in any computation; after such stabilization, it follows by self-stabilization of the update protocol that in $O(d)$ rounds, every image of a_p is also accurate. Then in one additional round statement U12 is executed at every processor, which implies stabilization to $(\forall p :: \neg f_p)$.

To show that the protocol stabilizes to $(\forall p :: a_p = 0)$, consider an arbitrary computation Φ. Observe that no statement of the code in Figure 8 can assign to a_p in the case that $a_p = 0$ is a precondition; in other words, $a_p = 0$ is locally stable. Therefore it suffices to show that eventually $a_p = 0$ is obtained for every p.

Suppose, heading for contradiction, that $a_p \neq 0$ holds for $p \in D$ throughout Φ, where D is some non-empty set of processors. Let Ω be a suffix of Φ in which no processor assigns $a_p := 0$. Thus statements U2, U4 and U6 are not executed by any processor in Ω. After $O(d)$ rounds of Ω, all a_p images accurately and permanently allow any processor p to test $a_r = 0$ for any known r; let Ψ be the suffix of Ω with this property. We now claim that maxb_p is bounded for any processor p in computation Ψ. Only two statements, U3 and U6 can increment an incarnation number, but U6 is eliminated from consideration by assumption. However if U3 is executed by p, then processor p is the only processor with $a_p \neq 0$. After $O(d)$ rounds of Ψ, all b_r images for $r \neq p$ are accurate. Thus U3 is executed a finite number of times at processor p. Let Υ be the suffix of Ψ in which U3 is not executed by any processor. Observe that $|D| = 1$ if the network's topology is connected. This follows because after $O(d)$ rounds of Υ, all b_p images are permanently accurate and if $a_p \neq 0$ and $a_r \neq 0$ for $p \neq r$, then one of statements U2 or U4 will execute, which is a contradiction. Hence there is at most one processor p satisfying $a_p = 1$ in any connected component in computation Υ. If $a_p = 1$ then eventually the image of a_p is broadcast by the update protocol and via the acknowledgement of U10, the predicate $(\forall q :: h_q[p] \neq 0)$ holds and p can detect this using images of the h_q fields. That is, either a processor q will acknowledge $a_p = 1$ or permanently retain $h_q[p] = 2$. In either case, eventually U5 is executed. And if $a_p = 2$ then by a similar argument, U6 is eventually enabled, but this contradicts the definition of D.

That f_p is a filter predicate is shown by the above contradition. It remains to show that convergence to $(\forall p :: \neg f_p)$ occurs in $O(d)$ rounds of any computation. In the case where a processor p is the only processor satisfying $a_p \neq 0$ within a connected component, the same update/acknowledge arguments given above show that in a constant number of broadcasts via the update mechanism, the protocol stabilizes to $a_p = 0$, which implies $O(d)$ convergence. In the case of numerous, competing processors satisfying $a_p \neq 0$, we appeal to arguments about maxb_p and statements U2 and U4 to conclude that a winner among the competing processors is obtained in $O(d)$ rounds. □

Lemma 10.2 The protocol is self-stabilizing and converges in $O(d+K)$ rounds where the protocol P self-stabilizes in $O(K)$ rounds.

Proof: Lemma 10.1 shows that after $O(d)$ rounds, all *freeze* bits are permanently *false*. Thus after $O(d)$ rounds the superstabilizer does not interfere with protocol P, which then self-stabilizes in $O(K)$ additional rounds. □

Lemma 10.3 The protocol of Figures 7 and 8 is continuously superstabilizing with superstabilization time $O(d)$.

Proof: To prove the lemma, the proof of Lemma 9.3 need only be modified so that safety conditions ("until" properties) account for larger incarnation numbers encountered during phase processing. The invariant is to show: for any set D of processors such that D represents a connected subgraph and f_p is *false* for all $p \in D$: all processors have equal incarnation numbers (b_p fields) and that $\sigma[D]$ is locally legitimate. This invariant must hold over a trajectory with any number of topology changes. The definition of $H[r]$ is that given in the proof of Lemma 9.3. With respect to state σ and processor r, let $leader.H[r]$ be the processor v with maximum identity such that $v \in H[r]$, v is incident on a topology change $\mathcal{E}$ prior to state σ, and b_v has the maximum incarnation number of any processor in $H[r]$. Definitions of Ph_1 and Ph_2 are modified to include incarnation numbers:

$$\begin{aligned} Ph_1 &\equiv a_v = 1 \ \wedge \ (\forall q : q \neq v : a_q = 0) \ \wedge \ (\forall q :: h_q[v] = 1 \ \wedge \ g_q[v] = b_v) \\ Ph_2 &\equiv a_v = 2 \ \wedge \ (\forall q : q \neq v : a_q = 0) \ \wedge \ (\forall q :: h_q[v] = 2 \ \wedge \ g_q[v] = b_v) \end{aligned}$$

Claims roughly equivalent to those given in the proof of Lemma 9.3 are:

1. For any processor p, $a_p = 1$ holds until either p observes Ph_1 or p observes $(a_q = 1 \ \wedge \ (\forall r :: h_r[q] = 1))$ for some processor $q > p$ and $b_q = b_p$, or p observes $b_q > b_p$ for some q satisfying $a_q \neq 0$.
2. For any processor p, f_p holds until v observes Ph_1.
3. For a constant K, following a topology change, there are no additional topology changes for Kd rounds, a state satisfying Ph_1 occurs; and if topology changes occur within Kd rounds, the leader v either changes identity or increases its incarnation number.
4. If there are no topology changes for Kd rounds following v observing Ph_1, then v observes Ph_2; and if there is a topology change within Kd rounds, then either leadership changes or incarnation number increases.
5. If v observes Ph_2, then either $a_p = 0$ for all p within Kd rounds, or the leadership changes due to a topology change.

The claims are seen to allow for leadership change during a trajectory due to topology change. We do not prove the claims in detail, since the arguments are the same as those made in the proof of Lemma 9.3 provided no leadership change occurs. The primary concern for showing correctness is that leadership changes are properly managed.

Suppose v is $leader.H[r]$ and a topology change occurs. At the instant of the change, v can be coordinating any one of the phases.

Phase one: If the topology change occurs at any state up to when v observes Ph_1, we can assert that $a_v = 1$ is acknowledged by any q only if $b_q = b_v$ and a snapshot is recorded; at such a state, there are two possibilities for incidence on the topology change: (1) the

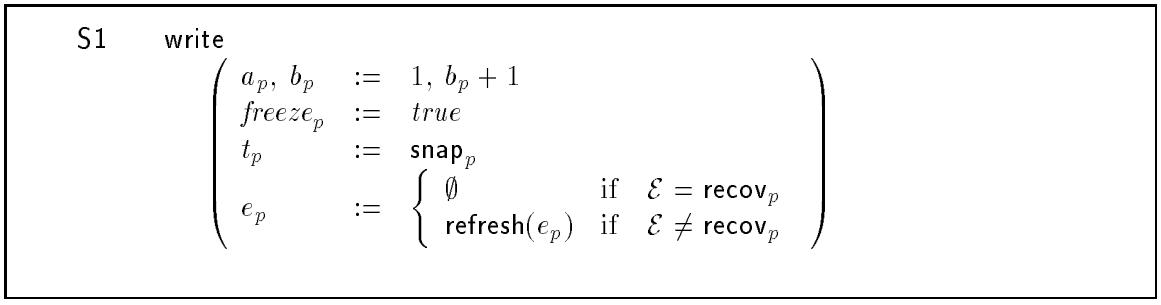

Figure 7: Continuous Superstabilizer: Interrupt Section for p.

processor incident on the change has not acknowledged $a_v = 1$ and can have $b_q \neq b_v$. If $b_q > b_v$ then a new leader is defined and we appeal to structural induction for safety properties of the new leader; if $b_q < b_v$ then the leadership does not change and the update algorithm insures that phase processing (made monotonic by incarnation numbers) continues for v; and if $b_q = b_v$ processors identities are used to decide leadership. A second possibility (2) is that the processor q incident on the change has acknowledged $a_v = 1$, but this implies $b_q > b_v$ as a result, and leadership changes properly.

Phase two: If the topology change occurs at any state following v's observance of Ph_1 (without leadership change) up to when v observes Ph_2, then all $q \in H[r]$ satisfy $b_q = b_v$. Any topology change incident on some processor in $H[r]$ results in a larger incarnation number and a new leader.

To show local legitimacy for any connected set D of unfrozen processors, consider the last snapshot distributed to processors in D in a trajectory. This snapshot is generated by a leader v that observes Ph_2, which implies stability of incarnation numbers between first and second phases; this stability guarantees that snapshot assembly is accurate and $\mathcal{F}(s_v)$ generates a new legitimate global state. This state is set at the instant $a_v := 2$ and f_p holds for all processors until v observes Ph_2; we conclude that the subset D of processors for which $\neg f_p$ holds is locally legitimate. □

11 Conclusions

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 be 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. Impetus for the research described in this paper

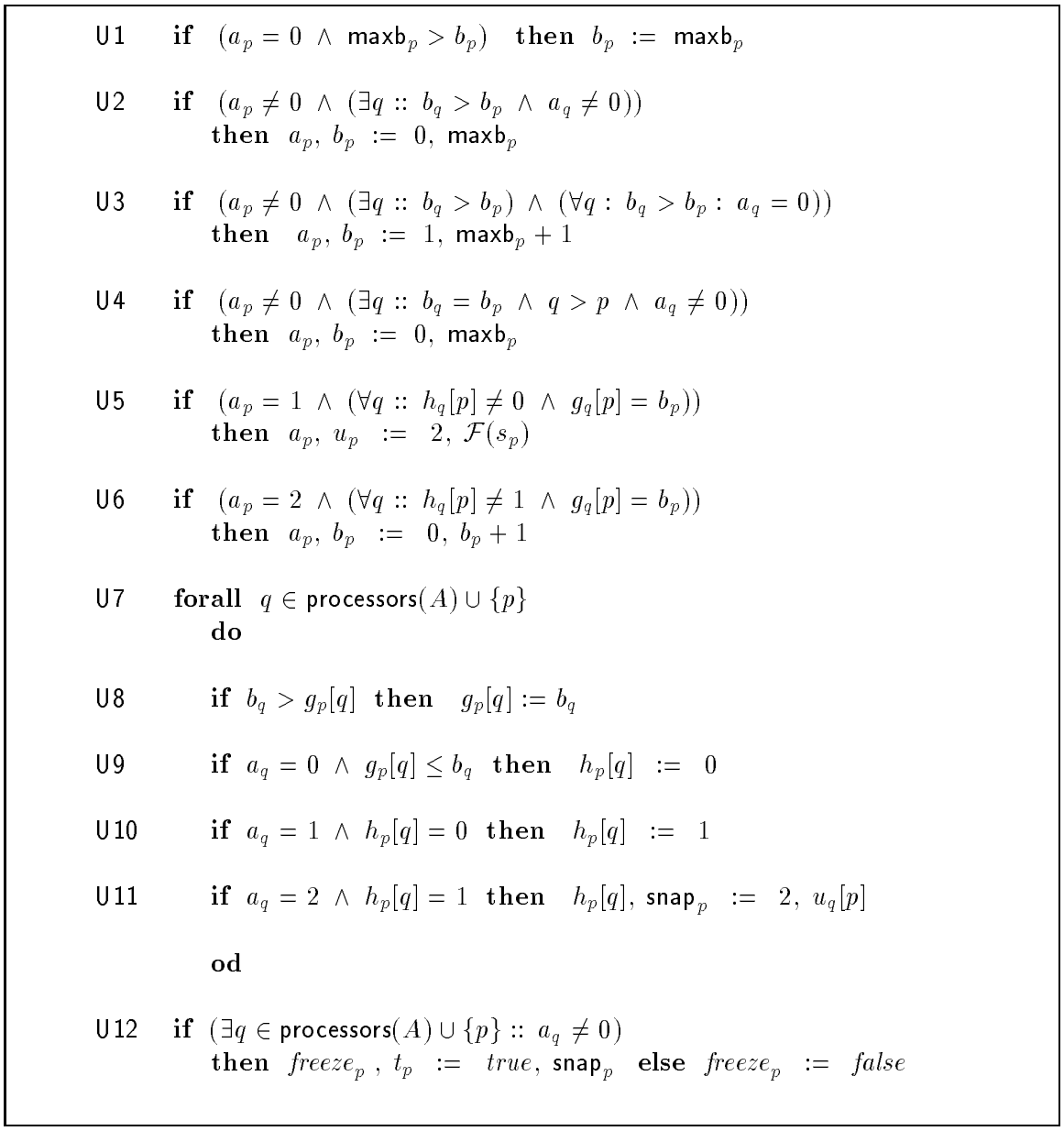

Figure 8: Continuous Superstabilizer: Update Extension for p.

is partly inspired by the thesis that the notion of self-stabilization has wider applicability than just fault-tolerance.

The general methods presented in Sections 9 and 10 demonstrate that superstabilization is, in principle, applicable to any self-stabilizing protocol. These methods, coupled with previous research devising general methods for making self-stabilizing protocols, are thus of potential use for a wide variety of network protocols. The general methods do not deliver optimum performance in all cases; the hand-crafted protocols of Sections 6 and 7 suggest that further research would be useful for specific problem domains. Our intention is that the general methods be regarded as existence proofs of superstabilizing protocols.

The examples of superstabilization in this paper happen to be protocols with $O(1)$ and $O(d)$ superstabilization time. However there can be problems, not examined in this paper, with superstabilization times between these extremes. Protocols with superstabilization time less than d also have qualitative advantages. For instance, our general methods can be improved not only by reducing superstabilization time for specific problems, but also by limiting the impact of change: instead of freezing the entire network following a change (which is the most conservative approach), it may be possible to freeze and reset only a portion of the network.

References

[AAG87] Y. Afek, B. Awerbuch and E. Gafni, "Applying Static Networks Protocols to Dynamic Networks," *Proc. of the 28th IEEE Symp. on Foundation of Computer Science* pp. 358-370, 1987.

[AB93] Y. Afek and G. M. Brown, "Self-Stabilization over Unreliable Communication Media," *Distributed Computing*, 7 pp. 27–34, 1993.

[ACK90] B. Awerbuch, I. Cidon and S. Kutten, "Communication-Optimal Maintenance of Replicated Information," *Proc. of the 31th IEEE Symp. on Foundation of Computer Science*, pp. 492-502, 1990.

[AEH92] E. Anagnostou, R. El-Yaniv, and V. Hadzilacos, "Memory Adaptive Self-Stabilizing Protocols," *Proc. of the 6th International Workshop on Distributed Algorithms*, pp. 203–220, 1992.

[AH93] E. Anagnostou and V. Hadzilacos, *Proc. of the 7th International Workshop on Distributed Algorithms*, 1993.

[AG90] A. Arora and M. G. Gouda, "Distributed Reset," *Proc. FST 10, Springer LNCS*, 472 pp. 316–331, 1990.

[AG92] A. Arora and M. G. Gouda, "Closure and convergence: A formulation of fault-tolerant computing," *Twenty-second Fault Tolerant Computing Symposium*, 1992.

[AGH90] B. Awerbuch, O. Goldreich and A. Herzberg, "A Quantitative Approach to Dynamic Networks," *Proc. of the 9th ACM Symp. on Principles of Distributed Computing*, pp. 189-203, 1990.

[AGR92] Y. Afek, E. Gafni and A. Rosen, "The Slide Mechanism with Applications in Dynamic Networks," *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pp. 35–46, 1992.

[AK93] S. Aggarwal and S. Kutten, "Time Optimal Self-Stabilizing Spanning Tree Algorithm," *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science,* 1993.

[AM92] B. Awerbuch and Y. Mansour, "An Efficient Topology Update Protocol for Dynamic Networks," *Proc. of the 6th International Workshop on Distributed Algorithms*, pp. 185-201, 1992.

[APV91] B. Awerbuch, B. Patt-Shamir and G. Varghese: "Self-Stabilization by Local Checking and Correction," *Proc. of the 32nd IEEE Symp. on Foundation of Computer Science* pp. 268–277, 1991.

[BGM93] J. E. Burns, M. G. Gouda, and R. E. Miller, "Stabilization and pseudo-stabilization", *Distributed Computing 7*, 1, pp. 35–42, 1993.

[BSW69] K. Barlett, R. Scantlebury, and P. Wilkinson. "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links," *CACM*, 12(5):260-261, May 1969.

[CL85] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, 3(1) pp. 63–75, 1985.

[CM88] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison–Wesley, 1988.

[Dij74] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *CACM* 17 pp. 643–644, 1974.

[Do92] S. Dolev, "Self-Stabilization of Dynamic Distributed Systems," *D.Sc. dissertation, Technion–Israel Institute of Technology*, June 1992.

[Do93] S. Dolev, "Optimal Time Self Stabilization in Dynamic Systems," *Proc. of the 7th International Workshop on Distributed Algorithms* (Springer-Verlag LNCS 725), pp. 160–173, September 1993.

[DIM93] S. Dolev, A. Israeli and S. Moran, "Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity," *Distributed Computing*, 7 pp. 3–16, 1993.

[DIM91] S. Dolev, A. Israeli and S. Moran, "Resource Bounds for Self Stabilizing Message Driven Protocols," *Proc. of the 10th Annual ACM Symp. on Principles of Distributed Computing*, pp. 281-293, 1991.

[DW93] S. Dolev and J. L. Welch, "Crash Resilient Communication in Dynamic Networks," *Proc. of the 7th International Workshop on Distributed Algorithms*, pp. 129-144, 1993.

[GH91] M. G. Gouda and T. Herman, "Adaptive Programming," *IEEE Trans. Soft. Eng.* 17 pp. 911–921, 1991.

[GM91] M. G. Gouda and N.J. Multari, "Stabilizing Communication Protocols," *IEEE Trans. Comp.* 40 pp. 448-458, 1991.

[GP93] A. S. Gopal and K. J. Perry, "Unifying Self-Stabilization And Fault-Tolerance," *Proc. of the 12nd Annual ACM Symp. on Principles of Distributed Computing*, pp. 195-206, 1993.

[KP93] S. Katz and K. J. Perry, "Self-Stabilizing Extensions for Message-Passing Systems", *Distributed Computing*, 7 pp. 17–26, 1993.

[La78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. of the ACM* 21,7, pp. 558-565, 1978.

[LL90] L. Lamport and N. Lynch, "Distributed Computing: Models and Methods," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen, Managing Editor, Elsevier, Amsterdam, 1990.

[SG89] J. Spinelli and R.G. Gallager, "Event Driven Topology Broadcast Without Sequence Numbers", *IEEE Transactions on Communication*, Vol. 37, No. 5, (1989) pp. 468-474.

12 Appendix: Update Protocol Proofs

Theorem 5 The update protocol of Figure 4 self-stabilizes in $O(d)$ rounds.

Proof: The the proof is organized as three claims.

Claim 1: 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 2: 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 same inductive argument presented for Claim 1.

Claim 3: 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 neighbour s the tuple $\langle q, , d \rangle$ and found no tuple with q as first component at a smaller distance. However, the tuple located at s having distance d represents the shortest distance to q by Claim 2. And since d bounds the maximum possible shortest path, by Claim 1 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 1–3 together imply that, following $d+1$ rounds, each processor correctly has a tuple for every other processor at distance d and that every tuple in an e-field correctly refers to a processor □

Theorem 6 The protocol of Figure 4 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 change $\mathcal{U}$ 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: 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 the type of change $\mathcal{E}$: 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 processor is added to the network; a topology change that decreases connectivity is labelled $-\mathcal{E}$.

For the case $+\mathcal{E}$ a technical lemma is needed: Lemma 12.2 shows for Φ that distances tracked in tuples do not increase during the computation, and that function *initseq* does not remove tuples during the course of the protocol's computation. To show impulse monotonicity, we assign one of two colours to each tuple in an e-register. Atomically with $+\mathcal{E}$ we colour all tuples white with the exception of $\langle , , 0\rangle$ tuples incident on $+\mathcal{E}$, which are coloured black. Then at each cycle of a processor in Φ, the colour of a $\langle , , 0\rangle$-tuple is black for processors incident on $+\mathcal{E}$ and white for other processors; the colour of a $\langle , , k\rangle$-tuple, $k \neq 0$, is inherited from the colour 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 neighbours 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 colour exactly once in computation Φ.

For the case $-\mathcal{E}$, the same colouring technique is used, with a different lemma: Lemma 12.1 shows for Φ that distances tracked in tuples to 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 colours to each tuple in an e-register. Atomically with $-\mathcal{E}$ we colour all tuples white with the exception of $\langle , , 0\rangle$ tuples incident on $-\mathcal{E}$, which are coloured black. Then at each cycle of a processor in Φ, the colour of a $\langle , , 0\rangle$-tuple is black for processors incident on $-\mathcal{E}$ and white for other processors; the colour of a $\langle , , k\rangle$-tuple, $k \neq 0$, is inherited from the colour 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 neighbours 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 colour exactly once in computation Φ. □

For the remaining lemmas of this subsection, σ, $\mathcal{E}$, $\mathcal{T}$, $\mathcal{U}$, and Φ are fixed as specified in Theorem 6. We label a topology change that increases connectivity as $+\mathcal{E}$, since either a link or processor is added to the network; a topology change that decreases connectivity is labelled $-\mathcal{E}$. 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 $dist_{\mathcal{U}}(p, q) \neq \infty$.

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.

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

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

(2) $\langle p,,\ell\rangle \in e_q \;\Rightarrow\; \ell \leq dist_{\mathcal{U}}(p,q)$

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

(4) $\langle p,,k\rangle \in e_q \odot \rho$ is based $\;\Rightarrow\; dist_{\mathcal{U}}(q,p) = k$

(5) $\langle p,,k\rangle \in e_q$ is based $\;\Rightarrow\; (\forall j :\; 0 \leq j \leq k :\; (\exists\langle r,,j\rangle \in e_q ::\; \langle r,,j\rangle \text{ is based}))$

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

Proof: Proof by induction on Φ.

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 (1) holds for λ by the assumption that σ satisfies $\mathcal{L}_{\mathcal{T}}$. Claims (2) and (4) 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 (3) and (6) are claims over pairs of distinct states and thus hold trivially in the initial state of Φ. Claim (5) follows from Claim (4), 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.

Induction Let $\delta = successor(\rho)$ and suppose that (1)–(6) hold for all states γ, $\gamma \preceq \rho$. Consider two cases for *adjust*: if $adjust(q,\rho,\delta)$ 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 (1)–(6) hold for δ by inheritance from ρ. The other possibility is that $adjust(q,\rho,\delta)$ holds for some processor q. In this case, the transition from ρ to δ writes

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 neighbours, which satisfy (1)–(6) by the induction hypothesis. To show that (1)–(6) hold for state δ, we consider the claims with respect to B, and then reason about *initseq*(B). The remainder of the induction considers tuples placed in B by steps of Φ preceding state δ.

Claim (1) holds for B because $\langle q, , 0\rangle \in e_q$ holds for any iteration of the loop in Figure 4 and by the induction hypothesis for (1), 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 (2) 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 (2) is assumed for r. Claim (2), the induction hypothesis (4), and the definition of a based tuple show that (4) holds for based tuples in B. Claim (5) holds by the induction hypothesis (5): based tuples in B are calculated upon neighbouring processor based tuples, which satisfy (5) by assumption; hence all of the neighbour's supporting based tuples (at smaller distances) are also input to forming tuples in B. Claims (3) and (6) are only concerned with tuples that change distance with respect to current distances in the e_q field. Claim (3) holds for tuples in B by the induction hypothesis for (1) and (3); tuples in B are calculated from neighbouring e-fields and tuples in these fields do not increase distance by any transition prior to state δ. Similarly, Claim (6) holds for B because any adjustment to a tuple follows from (possibly multiple) changes in neighbouring e-fields; by hypothesis (6), each such change to an e-field adjusts all maxlow tuples, which then by (1)–(2) and (4) remain constant thereafter.

Thus (1)–(6) have been established for B prior to the writing of *initseq*(B) at state δ. It only remains to show that no reachable tuple is removed from B by the application of *initseq*. This is argued by contradiction. Suppose a reachable tuple $\langle p, , m\rangle \in B$ is discarded by *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 (3). It follows that such a gap is the result of increasing the distance of some tuple(s). Yet (6) implies that the maximum-distance reachable tuple resulting from an increase yields a based tuple; (5) then implies the existance of tuples at all lesser distances in B, which contradicts the assumption of a gap. □

Lemma 12.2 For event $+\mathcal{E}$, for all processors p and q satisfying $dist_{\mathcal{U}}(p, q) \neq \infty$, the following claims hold:

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

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

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

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

Proof: Proof 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:

$$(\dagger)\ \ (\forall p, k ::\ \ dist_{\mathcal{U}}(p, z) = k \neq \infty\ \ \Rightarrow\ \ \langle p, , k\rangle \in e_z)$$

In addition to (1)-(4), we add (5) to the list of claims to prove invariant in the computation Φ:

$$(5)\ \ \langle z, , k\rangle \in e_q\ \ \Rightarrow\ \ (\forall r, \ell ::\ \ dist_{\mathcal{U}}(z, r) = \ell \neq \infty\ \ \Rightarrow\ \ (\exists \langle r, , m\rangle \in e_q ::\ \ m \leq k + \ell))$$

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 $(\dagger)$. State λ forms the induction's basis. Claim (1) 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 $(\dagger)$ directly implies (1) for processor z. Claims (2) and (3) 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 (4) holds trivially for λ since σ satisfies $\mathcal{L}_{\mathcal{T}}$, and (5) holds because no processor reads any tuple from e_z prior to state λ.

Induction Let $\delta = successor(\rho)$ and suppose that (1)–(5) hold for all states γ, $\gamma \preceq \rho$. Consider two cases for *adjust*: if $adjust(q, \rho, \delta)$ 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 (1)–(5) hold for δ by inheritance from ρ. The other possibility is that $adjust(q, \rho, \delta)$ holds for some processor q. In this case, the transition from ρ to δ writes $initseq(B)$ 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 neighbours, which satisfy (1)–(5) by the induction hypothesis. To show that (1)–(5) hold for state δ, we consider the claims with respect to B, and then reason about $initseq(B)$. The remainder of the induction considers tuples placed in B by steps of Φ preceding state δ.

Claim (1) 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 (1) by the induction hypothesis. Similarly, (4) and (5) follow by appealing to the induction hypothesis for the contents of some neighbouring processor's e-field. To show (3), 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 (3), 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 (2) 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

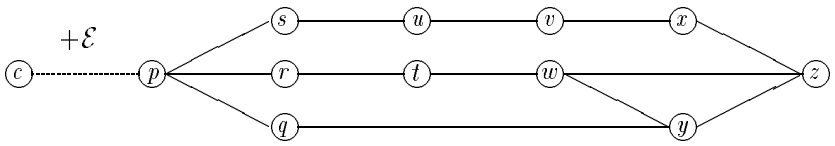

Figure 9: Network for Counterexample.

calculation of $\langle p, , m\rangle$ is based on neighbouring e-fields, all of whose tuples satisfy (2) by hypothesis, we conclude that (2) holds for B.

Thus (1)–(5) have been established for B prior to the writing of $initseq(B)$ at state δ. It only remains to show that no reachable tuple is removed from B by the application of *initseq*. This is argued by contradiction. Suppose a reachable tuple $\langle p, , m\rangle \in B$ is discarded by *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 (2). It follows that such a gap is the result of decreasing the distance of some tuple(s). This situation leads to the claim:

(6) $(\forall \langle r, , j\rangle \in B : \ j < \ell \ \wedge \ r \neq z : \ dist_{\mathcal{T}}(r, p) = \infty)$

Claim (6) follows from (4): on one hand, if $dist_{\mathcal{T}}(r, p) < (m - j)$ 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 $dist_{\mathcal{T}}(r, p) \geq (m - j)$ holds for every tuple $\langle r, , j\rangle \in B$, $j < \ell$, then tuples at distances $m, (m - 1), \ldots$ are by (4) present in B and there is no gap at distance ℓ. As a consequence of (6), there is some tuple $\langle p, , m\rangle \in B$ for which $dist_{\mathcal{T}}(q, p) = \infty$. Therefore $\langle p, , m\rangle \in B$ holds because some neighbouring 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 neighbour of z, call it s, so that $dist_{\mathcal{T}} = (q, s) = dist_{\mathcal{U}}(q, s)$, which by (1), (3) 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 (5) the existance of a gap is contradicted. □

Theorem 7 The update protocol of Figure 4 does not satisfy dynamic monotonicity.

Proof: The proof is by counter-example. Our counter-example is stronger than needed to disprove dynamic montonicity — the counter-example shows that even using the acknowledgement mechanism given in Section 9, dynamic monotonicity is not be acheived. Moreover, the counter-example uses only a single topology change at a legitimate state for the update protocol followed by only one additional change to a register field. Thus even a slight weakening of the impulse monotonicity property does not hold for the update

protocol. Instead of explaining the counter-example in terms of an x_p field, we use the terminology of Section 9 and use a_p and b_p subfields.

The counter-example consists of the following scenario. Topology $\mathcal{T}$ is given, as partly shown in Figure 9; the link c–p is not present in $\mathcal{T}$ and each node $i \in \{p, q, r, s, t, u, v, w, x, y, z\}$ satisfies $mdist_{\mathcal{T}}(i, c) > 50$ (this larger distance is realized through additional links and processors not shown in the figure). Processor c has boolean variables a_c and b_c which are broadcast via the update protocol to all other processors. The scenario begins in a legitimate state δ for the update protocol, in which all processors have the knowledge that $(a_c \wedge b_c)$ holds. We now consider a topology change $+\mathcal{E}$ that adds the link c–p, accompanied atomically by the change $a_c :=$ *false*. The subsequent computation for the counterexample is:

1. Processors p, s, u, v, x, and z execute in sequence one cycle each. As a result, processors have tuples for c as follows:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1			2		3	4		5		6
a_c	¬			¬		¬	¬		¬		¬
b_c											

   (blank table entries indicate that the processors has values from state δ.)

2. Processors x, v, u, s, p, and c execute in sequence one cycle each. As a result, processor c has updated distances to these nodes and also receives acknowledgement that these processors have "seen" that $\neg a_c$ holds.

3. Processors r, t, w, and y execute in sequence one cycle each. As a result, processors have tuples for c as follows:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1		2	2	3	3	4	4	5	5	6
a_c	¬		¬	¬	¬	¬	¬	¬	¬	¬	¬
b_c											

4. Processors w, t, r, p, and c execute in sequence one cycle each. As a result, processor c has updated distances to these nodes and also receives acknowledgement that these processors have seen that $\neg a_c$ holds.

5. Processors q executes a cycle, with the result:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1	2	2	2	3	3	4	4	5	5	6
a_c	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬
b_c											

6. Processors p and c execute in sequence one cycle each. As a result, processor c has an updated distance to q and now has collected acknowlegements from every processor that $\neg a_c$ holds.

7. Processor c writes $b_c := \mathit{false}$ into its register.

8. Processors p, s, u, v, x, r, t, and w execute in sequence one cycle each. Distances do not change, but the updated value of b_c is propagated:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1	2	2	2	3	3	4	4	5	5	6
a_c	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬
b_c	¬		¬	¬	¬	¬	¬	¬	¬		

9. Processor z executes a cycle, obtaining a smaller distance to c and thereby also updating its images of a_c and b_c, with the result:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1	2	2	2	3	3	4	4	5	5	5
a_c	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬
b_c	¬		¬	¬	¬	¬	¬	¬	¬		¬

10. Processor y executes a cycle, obtaining an updated distance from q and also copies the b_c value from q:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1	2	2	2	3	3	4	4	5	3	5
a_c	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬
b_c	¬		¬	¬	¬	¬	¬	¬	¬		¬

11. Processor z executes a cycle, obtaining a smaller distance to c from y, copying its images of a_c and b_c from y, with the result:

processor	p	q	r	s	t	u	v	w	x	y	z
$dist(c)$	1	2	2	2	3	3	4	4	5	3	4
a_c	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬	¬
b_c	¬		¬	¬	¬	¬	¬	¬	¬		

Thus a computation exists in which processor z obtains the sequence of b_c values [*true, false, true*] whereas the sequence of values at processor c is [*true, false*] for b_c. Dynamic monotonicity is therefore violated. □

13 Appendix: Message-based Implementation

Sections 2–5 propose a shared-register model with atomic execution of interrupt statements. The results are, however, intended for an asynchronous message-passing model. This section sketches the constructions needed to implement the shared-registermodel in terms of a reasonable message-passing system. Figure 10 presents a schematic view of a layered construction for the shared-register model. Each layer is a self-stabilizing protocol to deliver services to the next higher layer. The overall construction is a fair composition of the layers.

At the lowest layer we have a system of asynchronously executing, uniquely named processors that communicate by sending and receiving messages via local, numbered ports.

At the second layer, a processor knows which of the ports are active and the names of processors at the opposite ends of channels attached to the active ports. Since the status of ports and channels are dynamic, time-outs and probabilistic methods are used to implement a protocol that simulates the second layer. Also at this layer, a processor is able to distinguish between a link recovery and a processor recovery associated with a port becoming active.

The third layer is a modification of the alternating bit protocol for bounded channel suggested by [GM91, AB93, DIM91]. These self-stabilizing protocols also use time-out. We omit the requirement for the master-slave setup for the two ends of a link by the following technique: A message m received by a processor p from a processor q has two fields m_1 and m_2. p process both fields and sends a message m' with two fields m'_1 and m'_2 to q. Processor p processes m_1 as the receiver in the bounded protocol of [AB93] to produce m'_2. p process m_2 as the sender of the bounded protocol of [AB93] to produce m'_1. Similarly, q act as the receiver for m'_1 and as the sender for m'_2 and alternate the order of its responds. Thus, we view the link as two undirected links in one p is the sender and q is the receiver and in the other q is the sender and p is the receiver. Eventually, in each virtual link there exist exactly one token that circulates from p to q and backwards.

The fourth layer implements link registers using the alternating bit protocol provided by the third layer. The implementation idea was first introduced in [DIM91] and [Do92] for the case of implementing link registers by message passing. Below we sketch how essentially the same idea can be used for implementing a shared register as specified in Section 2. The fifth and sixth layers are discussed in earlier sections of this paper.

The heart of the implementation of the fourth layer is the simulation of virtual read and virtual write operations. Every processor has a local variable *virtual-register* that represents its register. A write operation is implemented by writing to the virtual-register. Every time a processor p receives a token at the virtual link in which p plays the receiver, p augments the token with the current value of its virtual-register. A read operation from a neighbour q is implemented by receiving a token from q in the virtual link in which p is the sender. Then receiving the *second* token from this virtual link and using the value augmented to it, constitutes the result of the read operation. Note that during the virtual read operation p continues to handle all the tokens arriving through every virtual link by augmenting the tokens with the value of its virtual-register.

To show that the implementation is correct one needs to show that the order of operations of every processor is preserved and the result of any virtual read from a register is a value that has been in that register during the virtual read operation. We map each virtual operation with real-time. The virtual write operation take place at the time the virtual-register is updated. The virtual read operation take place at the time the second token is initiated at the neighbour. Obviously, there is an execution in the shared register model that would have the write operation at the real-time of the virtual write operations

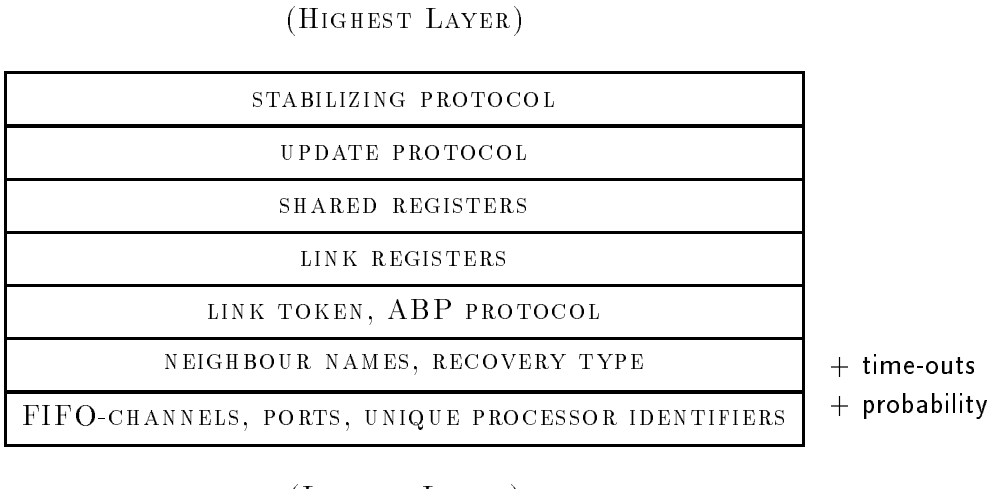

Figure 10: Schematic of Layered Register Construction.

and read operations at the real-time of the virtual read operations. This shared register execution is equivalent to that required by the model.