Fault-Containing Self-Stabilizing Algorithms

Sukumar Ghosh* Arobinda Gupta Ted Herman Sriram V. Pemmaraju
Department of Computer Science
The University of Iowa
Iowa City, IA 52242
{ghosh,agupta,herman,sriram}@cs.uiowa.edu

Abstract. Self-stabilization provides a non-masking approach to fault tolerance. Given this fact, one would hope that in a self-stabilizing system, the amount of disruption caused by a fault is proportional to the severity of the fault. However, this is not true for many self-stabilizing systems. Our paper addresses this weakness of distributed self-stabilizing systems by introducing the notion of *fault containment*. Informally, a fault-containing self-stabilizing algorithm is one that *contains* the effects of limited transient faults while retaining the property of self-stabilization. The paper begins with a formal framework for specifying and evaluating fault-containing self-stabilizing protocols. Then, it is shown that self-stabilization and fault containment are goals that can conflict. For example, it is shown that imposing a $O(1)$ bound on the worst case recovery time from a 1-faulty state necessitates added overhead for stabilization: for some tasks, the $O(1)$ recovery time implies stabilization time cannot be within $O(1)$ rounds from the optimum value. The paper then presents a transformer $\mathcal{T}$ that maps any non-reactive self-stabilizing algorithm P into an equivalent fault-containing self-stabilizing algorithm P_f that can repair any 1-faulty state in $O(1)$ time with $O(1)$ space overhead. This transformation is based on a novel stabilizing timer paradigm that significantly simplifies the task of fault containment. The paper concludes by generalizing the transformer $\mathcal{T}$ into a parameterized transformer $\mathcal{T}(k)$ such that for varying k we obtain varying performance measures for P_f.

1 Introduction

A self-stabilizing algorithm converges to some predefined set of *legitimate states* no matter what it has for its initial state. Due to this property, self-stabilizing algorithms provide means for tolerating transient faults. However, in the general area of fault tolerance, self-stabilization is classified as a non-masking approach [3], since the users of a stabilizing system can observe disrupted behavior while the system recovers from a transient fault. Given a non-masking fault-tolerant system, one would hope that the level of disruption observable by users be proportional to the severity of the transient fault causing the disruption. Unfortunately, many stabilizing systems do not have this property: in some cases even a single bit corruption can lead to an observable state change in all processes. Our current work addresses this particular weakness of distributed self-stabilizing algorithms by introducing the notion of *fault containment*. Informally, a fault-containing self-stabilizing algorithm is one that *contains* the effects of limited transient faults while retaining the property of self-stabilization.

The problem of containing the effects of limited transient faults is rapidly assuming importance for two reasons: (1) the dramatic growth in network sizes and (2) the fact that in practice, a transient fault usually corrupts a small number of components. For example, consider a broadcasting protocol that uses a spanning tree computed by an underlying self-stabilizing protocol. A transient fault at a single process, say i, that corrupts the spanning tree information local to i might contaminate a large portion of the system, thereby significantly and adversely affecting the broadcasting protocol. The damage to the broadcasting protocol could be in terms of lost messages or a large number of unnecessary messages. So our goal is to tightly contain the effects of limited transient faults. What "tightly" exactly means depends on the context and the application. For example in one context, tight fault containment could mean that observable disruption of the

*This author's research was supported in part by NSF under grant CCR-9402050.

PODC'96, Philadelphia PA, USA
© 1996 ACM 0-89791-800-2/96/05..$3.50

system state exists for at most $O(1)$ time; in another context tight fault containment could mean that during recovery from a 1-faulty state only processes within $O(1)$ distance from the faulty process are allowed to make observable changes in their local state.

An Example

To identify and emphasize the difficulties of combining fault containment with self-stabilization, we present a small example. The example is a simple stabilizing algorithm [6] to construct a spanning tree of a network of n processes. A particular process r is designated as the root of the spanning tree to be constructed. Each process i, $i \neq r$, has two variables $\langle p, \ell \rangle$, where p identifies i's parent in the tree and ℓ denotes the distance in the tree between i and r. The process r has a single variable ℓ whose value is set to 0. In the spanning tree algorithm, each non-root process perpetually checks its variables and adjusts them by applying one of two rules: (1) if $\ell < n$, then ℓ should be made one greater than that of its parent's ℓ variable; (2) if $\ell \geq n$ then p should be set to a new parent. Figure 1(a) shows a legitimate state for a small network (dashed lines are non-tree edges). The value of the variable ℓ for each process is shown beside the node and the parent variable p is shown by the arrow. At a legitimate state, only rule (1) applies and even when applied, it does not change the state of the process.

We now consider a single-process transient fault that occurs in the legitimate state shown in Figure 1(a). The transient fault changes the distance variable ℓ at process x and the resulting state is shown in Figure 1(b). In this state, rule (1) is applicable at both x and y. If this rule executes first at x, the global state is instantly repaired; however if y executes first, then every descendant of y could adjust its distance variable ℓ as well, before the system returns to a legitimate state. This example illustrates that optimal fault containment might be achievable by proper scheduling of the actions of an algorithm.

Figure 1(c) shows another example of a state obtained by a single-process transient fault at the state shown in Figure 1(a). This fault causes process x to change its parent variable p from w to z and the result is a cycle.

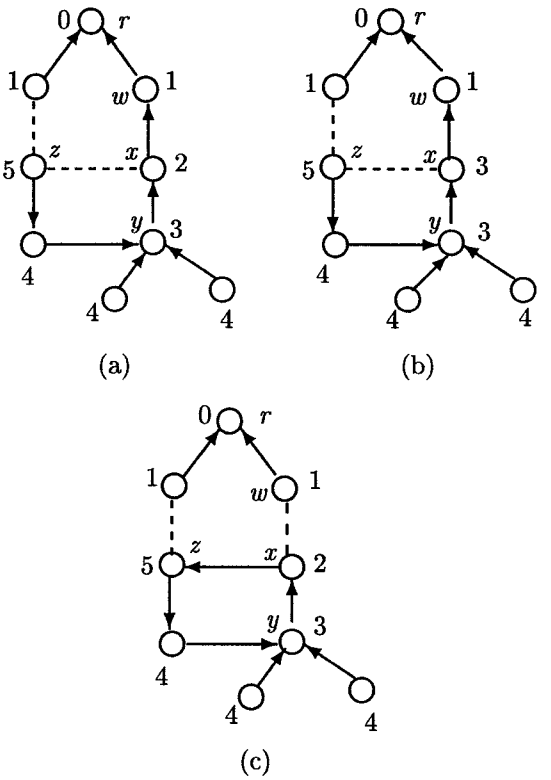

Figure 1: An example showing the difficulty of containing a fault

A legitimate state will be obtained after a number of rule executions by all processes in the cycle. Intuitively, one might hope that a transient fault at a single process could be repaired by actions only at that process, but this example shows stabilizing protocols may not have actions capable of such local repair. In particular, to recover from the 1-faulty global state shown in Figure 1(c) we would like x to reset its parent variable back to w. However neither of the two rules in the protocol allow x to take such an action. To overcome this deficiency, one might attempt to add actions explicitly for local repair following a single-process transient fault. However this leads to other dangers. Processes cannot decide based on local information, whether to execute the original protocol's actions or to execute the newly added local repair actions. Executing the local repair actions at the wrong time might undo some of the progress made by the original algorithm. (An account of how this spanning tree algorithm can be made fault-containing is given in [8].)

The spanning tree example points out the essential dif-

ficulty of introducing fault containment into stabilizing algorithms: based on local knowledge, processes cannot unambiguously decide the extent of a fault. One implication of this is that there is a thin line between fault containment and deadlock: on one hand, if a process chooses not to move the system might be deadlocked; on the other hand, if it chooses to move, then the system might not be fault-containing. A solution to this problem could be to extend the states of processes by adding auxiliary information so that each process knows more. But there is danger here as well, since a transient fault could corrupt the auxiliary information and lead to an equally unreliable local state.

Related Work

Of late, there has been growing interest among researchers in constructing protocols that are not only self-stabilizing, but also provide certain guarantees during convergence from certain states. The paper [11] proposes a stabilizing algorithm for a maximum flow tree in a network; the algorithm allows arc capacities to change and ensures that a tree is maintained during convergence to new flow values. Dolev and Herman [7] construct self-stabilizing protocols that guarantee safe convergence from states that arise from legitimate states perturbed by limited topology changes. We are aware of three examples of self-stabilizing algorithms that guarantee rapid convergence from states that arise from legitimate states due to single-process faults: [8] solves a spanning tree construction problem; [9] solves a leader election problem; [12] solves a mutual exclusion problem.

Fault containment is by no means limited to self-stabilization, and has been cited as a technique for fault tolerance [1, 15]. Kutten and Peleg [13] introduce the notion of fault mending, which addresses the issue of relating repair time to fault severity; their method is based on state replication and is self-stabilizing only in a synchronous system. It should be noted that one of our transformation techniques achieves $O(1)$ containment time without state replication, so that only $O(1)$ space overhead is needed.

Contributions

Our first contribution lies in formalizing the notion of fault containment within the context of self-stabilization. Based on this, we identify several ways of measuring the performance of fault-containing self-stabilizing algorithms. In this paper, we focus on single-process faults, but our definitions extend to multi-process faults also. We assume that a single-process fault can corrupt all of the variables in any one process. Our second contribution lies in identifying conflicts between the goals of fault containment and self-stabilization. For example, for the topology update problem we show that it is impossible to construct a fault-containing self-stabilizing protocol that optimizes both the worst case time to recover from a 1-faulty state as well as the worst case stabilization time. Our third contribution is a transformer $\mathcal{T}$ that converts a non-reactive self-stabilizing algorithm P into an equivalent fault-containing self-stabilizing algorithm P_f that repairs a 1-faulty state in $O(1)$ time with $O(1)$ space overhead. We use a novel timer-based technique to achieve this. We also generalize the transformer $\mathcal{T}$ into a parameterized transformer $\mathcal{T}(k)$ such that varying the parameter k changes the performance of the output algorithm P_f.

2 Fault Containment and Self-Stabilization

Let P_f be a self-stabilizing algorithm that stabilizes to a predicate L_f defined on the set of global states of P_f. Thus L_f identifies the set of legitimate states of P_f. Write each global state of P_f as an ordered pair $\langle p, s \rangle$, where p denotes the *primary* portion of the global state and s denotes the *secondary* portion of the global state. The secondary portion s may possibly be empty. The motivation behind this partitioning is that often the output of a self-stabilizing algorithm is represented by some variables, while the rest of the variables help in stabilization (and fault containment). So we denote that portion of the global state that is the output of the protocol and is of interest to the user as the primary portion; the rest we call the secondary portion. For example, in the spanning tree protocol described in Section 1, the primary portion can be thought of as the set of parent variables p and the secondary portion can be thought of as the set of distance variables

ℓ. Since the spanning tree constructed by the protocol can be completely represented by the parent variables, the role of the distance variables is to help in stabilization. Let L be a predicate defined on the set of global states of P_f such that:

(1) For any global state $\langle p, s \rangle$, $\quad L_f(p,s) \Rightarrow L(p,s)$.
(2) L is closed under execution of P_f (in any execution, once L is true, it remains true).
(3) L depends only on the primary portion of the state: $\quad (\forall p, s, s' : L(p,s) \equiv L(p,s'))$.

The motivation behind introducing the predicate L and defining it as above is as follows. Consider a legitimate state $\langle p, s \rangle$ for P_f. In such a state the predicate L_f is satisfied and as a result the predicate L is also satisfied. Suppose the system suffers a single-process fault and its state ceases to satisfy L (and therefore L_f); subsequently, with the help of the secondary portion of the global state, P_f quickly restores the system back to a state satisfying L. Furthermore, once L is satisfied, it remains satisfied independent of any changes being made to the secondary portion of the state. To any user of P_f, it will appear as though the single-process fault has been contained and quickly repaired. However, it is quite possible that many more state changes are needed before the entire global state, containing both the secondary and the primary portions, is restored to legitimacy. Only when this happens do we have a state that satisfies L_f. Note that if a single-process fault occurs in a state that satisfies $L \wedge \neg L_f$, the system provides no guarantee (beyond worst case stabilization time) as to how quickly this fault will get repaired. These observations motivate the following definition of fault containment in the context of self-stabilization.

Definition. Let a *1-faulty* state of P_f be a state obtained from a legitimate state of P_f by perturbing the local state of a single process. The protocol P_f is *fault-containing* with respect to the predicate L if from any 1-faulty state of P_f, the system reaches a state satisfying L in at most $O(1)$ time.

In the rest of this paper, we will simply refer to P_f as a fault-containing self-stabilizing protocol and not refer to L explicitly unless there is a possibility of confusion.

Depending on the application, alternate definitions of fault containment may be more appropriate. For example, a user might want to contain a fault not in time, but in the number of processes contaminated by the fault. In another case, the user might be interested in containing the effects of not one, but k faults, for some fixed $k \geq 1$. Given these alternate and plausible definitions, we have chosen a definition that simplifies exposition, and yet illustrates the phenomena we consider important.

Measures of Performance

Since we are interested in stabilization as well as fault-containment, we define two sets of measures for evaluating the performance of a self-stabilizing protocol P_f: one set to measure its stabilization property and the other set to measure its fault containment property.

Measures for Stabilization:

stabilization time: the worst case time to reach a legitimate state from an arbitrary initial state.

stabilization space: the maximum space used by any process in a legitimate state. Our measurement of space at a legitimate state (as opposed to space used during convergence to a legitimate state) is motivated by two previous papers on self-stabilization [2, 16].

Measures for Fault-containment:

containment time: the worst case time for P_f to reach a state satisfying L from any 1-faulty state. This is a measure of how quickly a 1-faulty state is observably corrected by P_f. Note that the containment time for any fault-containing self-stabilizing algorithm is $O(1)$.

contamination number: the worst case number of processes that change the primary portion of their state during recovery from a 1-faulty state to a state satisfying L. This is a measure of how many processes make observable changes to their local states, before the 1-faulty state is repaired. Ideally, we would like only the faulty process to change the primary portion of its state, and in such a case the contamination number would be 1.

fault gap: the worst case time to reach a state satisfying L_f from any 1-faulty state. Note that this is

at least as large as the containment time and in general can be much larger. The fault-gap can be thought of as a lower bound on how far apart in time two single-process faults have to be if they are to repaired in $O(1)$ time by a fault-containing self-stabilizing algorithm. This is because a fault-containing self-stabilizing algorithm provides $O(1)$ time repair guarantee *only* if a single-process fault occurs in a legitimate state, that is, a state satisfying L_f.

containment space: the maximum size of the secondary portion of the local state of any process at a legitimate state. This is a measure of the space overhead per process for fault containment.

3 The Transformation Problem

Let P_f be a fault-containing self-stabilizing protocol, as defined in Section 2. Let the *primary state space* of P_f be the set of all p such that $\langle p, s\rangle$ is a global state of P_f.

Definition. The *Transformation Problem* is the following: Construct a transformer $\mathcal{T}$ that inputs a self-stabilizing protocol P, where P stabilizes to a predicate L', and outputs a fault-containing self-stabilizing protocol P_f such that: (1) the global state space of P is equal to the primary state space of P_f and (2) $(\forall p, s : \ L'(p) \equiv L(p, s))$, i.e., the predicate L is a natural extension of L' to the state space of P_f.

Note that since P is self-stabilizing, the global state of P may be partitioned into a primary portion and a secondary portion. In fact, P may itself be fault-containing. Independent of whether P is fault-containing or not, the transformer $\mathcal{T}$ constructs a protocol P_f that is essentially identical to P, except that P_f comes with a guarantee of fault-containment. The transformer maps each global state of P (including both its primary and secondary states) into the primary state of P_f. The secondary state of P_f contain some auxiliary variables that it uses for fault-containment. In [8], the authors modify the spanning tree algorithm described in Section 1, by adding auxiliary variables (and corresponding program statements) so as to make it fault-containing. The primary portion of the modified algorithm contains the variables p as well as ℓ, while the secondary portion contains the new auxiliary variables.

4 Model of Computation

We consider a network of processes connected by bidirectional channels according to an arbitrary topology. For a process i, N_i denotes the set of neighbors of i in the network. We assume the locally shared memory model of communication. Thus, each process i has a finite set of *local variables* such that the variables at a process i can be read by i or any neighbors of i, but can be modified by i only. Each process has a program and processes execute their programs asynchronously. We assume that the program of each process i consists of a finite set of guarded statements of the form $G \rightarrow A$, where G is a boolean predicate involving the local variables of i and the local variables of its neighbors, and A is an assignment that modifies the local variables in i (expressions in A can refer to variables of i and variables of its neighbors). The *action* A is executed only if the corresponding guard G evaluates to true, in which case we say guard G is enabled.

We assume that each guarded statement is executed atomically (however our results may hold with finer levels of atomicity). The atomic execution by process i of a statement with an enabled guard is called a *program step* of i and can be represented by a pair of global states. A system execution is a sequence of global states such that any pair of consecutive states in the sequence correspond to some program step. In the sequel, we assume that the action of any guarded statement falsifies its guard; hence no program step occurs twice in succession in an execution (we disallow stuttering executions). We also assume, for the remainder of the paper, that all programs are non-reactive [14], that is, we are dealing with algorithms that have finite executions (no guard is enabled at the final state).

We measure time in *rounds*. Informally, a round is completed when each process has evaluated all its guards at least once, and every process whose guards are evaluated to be true has executed the action corresponding to at least one enabled guard. More precisely, let s_1 denote the initial state of an execution sequence. For any $r \geq 1$, let s_r be the global state at the beginning

of round r. Then, round r is the minimal contiguous subsequence of states in the execution sequence, starting in state s_r, such that for every process i, either there exists a state in round r in which i has no enabled guards or round r contains a program step of i. The state immediately after round r is denoted s_{r+1}.

5 Impossibility of Optimum Fault Gap and Stabilization Time

Our goal is not to build any transformer, but to build an "optimal" transformer. Since we have defined fault containment in terms of recovery time from a 1-faulty state, a natural question is to investigate the worst case stabilization time from a 1-faulty state. As suggested in Section 1, there appear to be intrinsic difficulties in attempting to optimize an algorithm's handling of 1-faulty states as well as worst case initial states. Our main result confirms this intuition.

Theorem 1 There exists no transformer that maps every self-stabilizing protocol P with stabilization time T into a fault-containing self-stabilizing protocol P_f such that: (i) the stabilization time of P_f is $T + a$ for some constant a and (ii) the fault gap for P_f is $O(1)$.

This theorem tells us that independent of any of the other measures of P_f, it is not possible for any transformation to optimize both stabilization time and fault gap in every case. In fact, there might be a precise trade-off between these two measures; we leave the exploration of this trade-off for future research. To prove Theorem 1 we start with the *topology update* problem on a linear array of n processes. A self-stabilizing protocol can solve this problem in n rounds [7]. So what happens when we impose the requirement of small fault gap on any algorithm that solves the topology update problem? The following theorem shows that if the fault gap is sublinear, then the overhead for stabilization cannot be constant.

Theorem 2 Let P_f be a self-stabilizing protocol solving the topology update problem. If the fault gap for P_f is $o(n)$, then the stabilization time for P_f exceeds $n + a$ for any given constant a.

Proof Sketch: The intuition is that an initial state σ for an array of processes can be constructed requiring at least $n - 2$ rounds for stabilization. There exists an execution, starting at σ, which appears locally to be in a 1-faulty state at a significant fraction of the $n - 2$ rounds. At each round in this execution, there is a process that is critical to the optimum progress toward stabilization — that process must propagate topology information; however, the state appears to the critical process to be 1-faulty in such a way that the process must first communicate with a neighbor before it can propagate the topology information. If the critical process does not engage in such communication, we can construct another state σ' that is 1-faulty but requires a linear fault gap for convergence to L_f. □

Theorem 2 implies there exists no transformer that can take an optimal self-stabilizing algorithm for the topology update problem and produce a protocol with constant time overhead for self-stabilization and constant fault gap. Hence, Theorem 1 follows as a corollary to Theorem 2. A theorem similar to Theorem 2 can be proved using, instead of fault gap, the number of processes that change state (primary or secondary) during convergence from a 1-faulty state. As a corollary of that theorem, we also have the following.

Theorem 3 There exists no transformer that maps every self-stabilizing protocol P with stabilizing time T into a fault-containing self-stabilizing protocol P_f such that: (i) the stabilization time of P_f is $T + a$ for some constant a and (ii) the number of processes that change state during convergence from a 1-faulty state is constant.

The results of this section *do not* rule out the possibility that a transformer can be used to make stabilization-time optimal protocols that are fault-containing, since containment time and fault gap have different meanings. Indeed, these impossibility theorems motivate the distinction between containment and stabilization.

6 Solution to the Transformation Problem

In this section we present a solution to the Transformation Problem. We construct a transformer $\mathcal{T}$ that takes

a non-reactive self-stabilizing protocol P and maps it into an equivalent non-reactive fault-containing self-stabilizing protocol P_f. To simplify presentation, we assume that in protocol P, the program of each process i consists of a single guarded statement $G_i \rightarrow A_i$. Note that in a non-reactive protocol, no guard is enabled in the final state. Hence, the existence of an enabled guard indicates an illegitimate global state. Thus, the protocols that we consider are locally checkable [5].

The main idea in constructing P_f from P is the following. View the protocol P_f as consisting of two protocols C and C' cascaded together such that C executes first followed by C'. C is a synchronous non-self-stabilizing protocol that performs the following task: if started in a 1-faulty state of P_f, C takes the system into a state satisfying L in $O(1)$ time. C' is the protocol P along with some additional bookkeeping operations necessary to establish the truth of L_f after L is true. Thus, when P_f is started in a 1-faulty state, C executes first and in $O(1)$ time repairs the fault and takes the system into a state satisfying L. Then C' executes and takes the system into a state satisfying L_f. If P_f is started in a state that is not 1-faulty, first C executes and does nothing useful and then C' executes and takes the system into a state satisfying L_f.

The implementation of the above idea depends on co-ordination between the two protocols C and C'. This co-ordination is provided by a *timer* protocol described in the following subsection. This timer protocol is similar to the `synchronizer` protocol [4], the main difference being that our timer is stabilizing and terminates when stabilization is completed.

6.1 Timer Protocol

The timer protocol maintains a timer variable t_i at each process i. This variable can take any integer value in the range $[0..M]$, where M is a positive integer whose value will be specified later. The timer variable t_i is *consistent* if its value differs by at most one from the value of the timer variable of each of its neighbors; otherwise t_i is said to be *inconsistent*. The goal of the timer protocol is to make the timer variables consistent with each other when they have a sufficiently large value and then decrement them without loosing consistency so that eventually all timer variables have the value 0.

do
[1] $raise(t_i)$ $\longrightarrow$ $t_i := M$
[2] $\Box$ $decrement(t_i)$ $\longrightarrow$ $t_i := t_i - 1$
od

Figure 2: Implementation of Timer. Program for process i.

Let n be the number of processes in the network. An implementation of the timer is shown in Figure 2. The protocol consists of two rules, one that sets the timer variable of a process to M and the other that decrements the timer variable. The timer variable of a process is set to M if either one of the following conditions is met:

(1) $t_i \neq M \wedge (\exists j \in N_i : (t_i - t_j > 1) \wedge (t_j < M - n))$
(2) $(\exists j \in N_i : t_j = M \wedge (t_i < M - n))$

Thus the predicate $raise(t_i)$, that appears in Rule `(1)` in Figure 2 is the disjunction of the two predicates specified above. Note that $raise(t_i)$ is true only when t_i is inconsistent.

The timer variable of a process is decremented if either one of the following conditions is met:

(1) $t_i > 0 \ \wedge \ (\forall j \in N_i : 0 \leq t_i - t_j \leq 1)$
(2) $(\forall j \in N_i : t_i \geq t_j \ \wedge \ (M - n \leq t_j))$

Thus the predicate $decrement(t_i)$, shown in Rule `(2)` in Figure 2 is the disjunction of the two predicates specified above. Note that both conditions require t_i to be no less than the timer value of any neighbor. Condition `(1)` requires that t_i be consistent, while Condition `(2)` allows t_i to be decremented even when it is inconsistent, provided that the neighbors of i have timer values in the range $[M - n..M]$. All guards in the timer protocol become false when all timer values are 0. The important feature of the timer protocol is that it provides synchronization between neighboring processes. In particular, the only way a timer value can come down to 0 is through decrement actions that are "synchronized" with neighbors.

Note that apparently, a simpler timer protocol could be constructed that sets the timer variable of a process

do

[1] $raise(t_i) \ \vee \ (t_i = 0 \ \wedge \ (\neg L_c \ \vee \ G_i))$
$\longrightarrow \ t_i := M$

[2] $\square$ $decrement(t_i)$
$\longrightarrow \ action(t_i); \ t_i := t_i - 1$

od

Figure 3: Protocol P_f based on Timer. Program for process i.

to M whenever it is inconsistent (irrespective of its current value), and decrements the timer variable only when it is consistent and is greater than or equal to the timer variables of each of its neighbors. However, an initial state can be constructed to show that this simple protocol livelocks when run on a network with cycles, and hence, the protocol is not self-stabilizing. The protocol presented here prevents livelock by ensuring that a timer value of M cannot circulate indefinitely in a cycle. Note that the maximum length of a cycle in a network of n processes is n.

6.2 The Protocol P_f

The variables of the protocol P_f are the variables of the protocol P (the self-stabilizing protocol input to the transformer), along with any additional variables used by protocol C (the synchronous non-self-stabilizing protocol that, when started in a 1-faulty state of P_f, establishes L in $O(1)$ time) and the timer variables. In a legitimate state of P_f, characterized by the predicate L_f, the variables of P are in a legitimate state, characterized by the predicate L, the timer variables are all equal to 0, and the additional variables of C are in a legitimate state. Let the predicate L_c identify states in which the variables of C are legitimate. Note that in a state satisfying L_c, what the values of the variables of C should be may depend on the values of the variables of P.

The protocol P_f is shown in Figure 3. Note that the only change to the timer protocol is the weakening of the first guard and the addition of an action to the second guarded statement. We now provide a high-level description of each of the two guarded statements in the protocol.

Statement 1. Suppose that a single-process fault occurs at a process i in a legitimate state of P_f. This fault triggers process i or one of its neighbors to set their timer variable to M. This acts as a signal that spreads across the entire network and all processes eventually set their timer variable to M. This process of signaling a 1-faulty state is implemented by the first guarded command in Figure 3. In case of arbitrary faults, multiple processes may set their timer variables to M independently. However, it can be shown that the timer variables of all processes become consistent with a sufficiently large value within a finite time. It is important for the correctness of protocol P_f that the timer variables of all processes become consistent and then decrement down to 0 maintaining consistency over a sufficiently large interval of the timers.

Statement 2. When a process is ready to decrement its timer, it does so, but before that, depending on the value of its timer variable, the process executes an action. This is implemented by the second guarded command in Figure 3. We now describe $action(t_i)$, which is the action performed by process i when it is ready to decrement its timer variable. Corresponding to different values of t_i, different actions are performed and these are described below. Choosing $action(t_i)$ appropriately for different values of t_i allows us to coordinate C and C', as desired. The timer variable range, $[1..M]$ is partitioned into the following subranges:

(1) $[M - c + 1..M]$. Here c is the worst case running time of C (in synchronous rounds). If t_i is in this range, then i executes an appropriate action of the protocol C. In particular, if the c rounds of the protocol C are consecutively numbered from 0 to $(c - 1)$, then i executes round m of C if t_i is equal to $M - m$, and then decrements t_i. It can be shown that starting from a 1-faulty state, t_i is decremented only when all neighbors of i has timer values equal to t_i or one less than t_i. Thus synchronizing the actions of C using the timer allows us the luxury of assuming that C is synchronous. This is because a process i executes an action of C only when all neighbors have completed executing actions from the previous synchronous round. Thus each process executes protocol C when its timer variable is in the range $[M - c + 1..M]$.

(2) $[b+1..M-a]$, for some $b \geq 1$, where $a = \max(c, n)$.

If t_i is in this range, then i executes the action A_i provided G_i holds. Thus each process executes protocol P when its timer variable is in the range $[b+1..M-a]$. Note that for a particular n, if $c \geq n$, then this subrange is contiguous to the subrange $[M-c+1..M]$ above. However, if $c < n$, then no action is taken if t_i is in the intermediate subrange $[M-n+1..M-c]$.

(3) $[2..b]$. If t_i is in this range, then some book-keeping operations are performed. These operations restore the variables of C to legitimacy and could have been performed in the range $[M-c+1..M]$, but it is easier to perform these operations after the variables in P have achieved legitimacy. Whether these book-keeping operations are really necessary, depends on the implementation of the protocol C.

(4) $[1..1]$: If t_i is in this range, then no action is taken.

The only restriction on the value of M is that it should be large enough to allow the protocol P to reach legitimacy when it executes in the range $[b+1..M-a]$, where $a = \max(c, n)$. Then, in case of a single fault, it can be shown that C executes first to establish L in constant time. Since the truth of L implies G_i is false for all i, no state change occurs when the timer is in the subrange $[b+1..M-c]$. The bookkeeping operations are then performed to restore the truth of L_c if necessary in the subrange $[2..b]$. In case of an arbitrary fault, the execution of P in the subrange $[b+1..M-a]$ establishes L.

6.3 Implementations of C

In this section, we sketch two implementations of C. Both implementations ensure that the contamination number of P_f is 1. However, the first implementation is simple, but space inefficient, while the second implementation is more complicated, but space-efficient.

Implementation 1. Without loss of generality, we will assume that the protocol P uses a single variable x_i at each process i. The protocol C uses an additional variable s for each process. The variable s is a vector, with one element $s[j]$ for each neighbor j of i. In a legitimate state, $s[j]$ contains a copy of x_j. C can be implemented in 3 rounds as follows. In round 1, a process i with more than one neighbor can correct x_i by consulting the variable s at each neighbor j. In round 3, a process i with exactly one neighbor can correct x_i in a similar manner. Note that a process with one neighbor has access to exactly one s variable and therefore can be fooled into correcting x_i even when it should not. This possibility is avoided by correcting any faulty s variables in round 2. The exact details of the implementation are omitted from this preliminary version. This implementation of C yields a containment space of $O(d \cdot x)$ for P_f, where d is the maximum degree of any node in the network and x is the maximum space used by protocol P at any process in a legitimate state.

Implementation 2. We now sketch a space-efficient implementation of C that uses temporary variables, and adds only $O(1)$ space overhead to each process when the system is in a legitimate state. So if we use this implementation of C, the containment space of P_f is $O(1)$.

Let NS_i denote the set of local states of i and its neighbors. The set NS_i is *inconsistent* if either G_i is true, or G_j is true, for some neighbor $j \in N_i$; otherwise NS_i is *consistent*. A *stabilizing change* is a state change at process i that will make NS_i consistent if it is inconsistent. If a single-process fault occurs at i, clearly there exists a stabilizing change at i. C can be implemented in 3 rounds as follows. In round 1 process i constructs a set X that contains all possible stabilizing changes at i. In round 2, each $j \in N_i$ constructs a set Y, $Y \subseteq X$, that contains all state changes of i in X that will make NS_j consistent. In round 3, process i makes a state change that is an element of Y in all neighbors. In order to construct X in round 1, process i needs to know the status of G_j for each neighbor j. This gathering of information can be implemented in an additional round. Note that in a legitimate state, the timer variable at a process can be encoded by a single bit (indicating if the value of the timer is 0 or not), thereby resulting in only $O(1)$ space overhead per process when the system is in a legitimate state.

The following theorem summarizes the construction presented in this section.

Theorem 4 Let P be a non-reactive self-stabilizing protocol with stabilization time T running on a network with diameter D. The transformer $\mathcal{T}$ takes as input P and produces as output a non-reactive fault-containing self-stabilizing protocol P_f with the following performance measures: (a) containment time $O(1)$, (b) contamination number 1, (c) fault-gap $O(T \cdot D)$, and (d) containment space $O(1)$.

6.4 A Spectrum of Transformations

So far, we have assumed that the range M is large enough for P to reach legitimacy when the processes execute P in the timer range $[b+1..M-a]$, where $a = \max(c, n)$. When M is large enough, the execution of P_f can be viewed as the execution of C followed by the execution of P. Now suppose that we shrink the value of M to a M/k. P_f is still a fault-containing self-stabilizing algorithm, except that in case of multi-process faults, its execution can now be viewed as alternate executions of C and P, k times. This may introduce the problem of intermediate executions of C undoing the progress made by P. However, this problem can be easily removed with a slight modification of our implementation. It can be shown that by increasing k we can decrease the fault-gap of P_f and in fact by choosing k large enough we can reduce the fault-gap of P_f to $O(n)$. Of course, there is an overhead to pay in terms of stabilization time that increases as k increases.

7 Extensions

There are various possible ways of extending this work. Some of the questions we consider important are:

(1) The transformer presented in this paper has a output P_f with a high fault-gap. Is it possible to build a transformer whose output has $O(1)$ fault-gap? We are currently investigating the design of such transformers.

(2) Is it possible to build a transformer whose output P_f has $O(1)$ fault-gap and $O(T+D)$ stabilization time, where T is the stabilization time of the input P and D is the diameter of the network. Our impossibility results in Section 5 rule out the possibility of an $O(1)$ fault-gap along with an $T+O(1)$ stabilization time, but do not rule out an affirmative answer to the above question.

(3) Is it possible to extend our results for single-process faults to the case of k-process faults for some fixed k?

Bibliography

[1] R. J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability, and Safety," *ACM Computing Surveys*, Vol. 22, 1990, pp. 35–68.

[2] E. Anagnostou, R. El-Yaniv, and V. Hadzilacos, "Memory Adaptive Self-stabilizing Protocols", *Sixth International Workshop on Distributed Algorithms*, Springer LNCS 647, 1992, pp. 203–220.

[3] A. Arora and M. G. Gouda, "Closure and Convergence: A Foundation for Fault-Tolerant Computing," *IEEE Transactions on Software Engineering*, Vol. 19, 1993, pp. 1015–1027.

[4] B. Awerbuch, "Complexity of Network Synchronization," *Journal of the ACM*, Vol. 32, 1985, pp. 804–823.

[5] B. Awerbuch and G. Varghese, "Self-Stabilization by Local Checking and Correction," *IEEE Symposium on Foundations of Computer Science*, 1991, pp. 268–277.

[6] N. S. Chen, H. P. Yu, and S. T. Huang, "A Self-Stabilizing Algorithm for Constructing a Spanning Tree," *Information Processing Letters*, Vol. 39, 1991, pp. 147–151.

[7] S. Dolev and T. Herman, "Superstabilizing Protocols for Dynamic Distributed Systems," *Second Workshop on Self-Stabilizing Systems*, 1995, pp. 3.1–3.15.

[8] S. Ghosh, A. Gupta, and S. V. Pemmaraju, "A Fault-containing Self-stabilizing Spanning Tree Algorithm," Technical Report TR-95-08, Department of Computer Science, The University of Iowa, Iowa City, 1995.

[9] S. Ghosh and A. Gupta, "An Exercise in Fault-containment: Leader Election on a Ring," submitted for publication.

[10] M. G. Gouda, "The Triumph and Tribulation of System Stabilization," *Ninth International Workshop on Distributed Algorithms*, Springer LNCS 972, 1995, pp. 1–18.

[11] M. G. Gouda and M. Schneider, "Maximum Flow Routing," *Second Workshop on Self-Stabilizing Systems*, 1995, pp. 2.1–2.13.

[12] T. Herman, "Superstabilizing Mutual Exclusion," *International Conference on Parallel and Distributed Systems*, Athens, GA., 1995.

[13] S. Kutten and D. Peleg, "Fault-Local Distributed Mending," *Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 20–27.

[14] Z. Manna and A. Pnueli, *The Temporal Aspects of Reactive and Concurrent Systems*, Springer Verlag, New York, 1992.

[15] V. C. Nelson, "Fault-tolerant Computing: Fundamental Concepts", *IEEE Computer*, Vol. 23, No. 7, July 1990, pp. 19-25.

[16] G. Parlati and M. Yung, "Non-Exploratory Self-Stabilization for Constant-Space Symmetry Breaking", Algorithms ESA 94, Springer LNCS 855, 1994, pp. 183–201.