Time-Adaptive Self Stabilization

Shay Kutten
Dept. of Industrial Engineering
The Technion
and
IBM T.J. Watson Research Center
kutten@ie.technion.ac.il

Boaz Patt-Shamir*
College of Computer Science
Northeastern University
boaz@ccs.neu.edu

Abstract

We study the scenario where a transient fault hit f of the n nodes of a distributed system by corrupting their state. We consider the basic problem of *persistent bit*, where the system is required to maintain a value in the face of transient failures by means of replication. We give an algorithm to recover the value quickly: the value of the bit is recovered at all nodes in $O(f)$ time units for any unknown value of $f < n/2$. Moreover, complete state quiescence occurs in $O(\mathbf{diam})$ time units, where **diam** denotes the diameter of the network. This means that the value persists indefinitely so long as any $f < n/2$ faults are followed by $\Omega(\mathbf{diam})$ fault-free time units. We prove lower bounds which show that both time bounds are asymptotically optimal.

Using the algorithm for persistent bit, we present a general transformer which takes a distributed non-reactive, non-stabilizing protocol $\mathcal{P}$, and produces a self-stabilizing protocol $\mathcal{P}'$ which solves the problem $\mathcal{P}$ solves, with the additional property that if the number of faults that hit the system after stabilization is f, for any unknown $f < n/2$, then the output of $\mathcal{P}'$ regains stability in $O(f)$ time units, and the state stabilizes in $O(\mathbf{diam})$ time units.

*Research supported by DARPA and Rome Laboratory under agreement F30602-96-0239.

1 Introduction

Self-stabilizing distributed systems enjoy a unique robustness property: they can recover from any kind of transient fault, provided that program code remains intact. One of the common methods to make a system self-stabilizing is the *reset* paradigm, which offers self-stabilization with low space and communication overhead, at the price of high stabilization time—$\Omega(\mathbf{diam})$ time units is a trivial lower bound, where **diam** denotes the network diameter. In this paper we study a different type of solutions we call *time-adaptive protocols*: these are protocols which recover from faults very quickly, typically at the cost of higher space and communication overhead. We believe that this end of the tradeoff spectrum is worthy of further exploration.

Let us elaborate on time-adaptivity a little. Ideally, we would like a system to recover from faults in the shortest possible time. Clearly, recovery time heavily depends on the severity of the transient fault. In particular, it is influenced by the number of processors whose state was corrupted by the fault. The notion of "faulty processors," however, requires a sound definition: for example, consider a system with n processors where one of the processors is not consistent with the others. Did a transient fault hit that outcast processor, or did the fault corrupt the state of all other $n-1$ processors in a consistent way? An operational approach [10] is to select the unknown answer to this question to be the definition. In this paper, we use a state-based measure (akin to Hamming distance used in error correcting codes, also used in [7]): the number of faults is defined to be the minimum number of processors whose state need be changed to attain a stable global state.

Note that according to this definition, a non-faulty node may become faulty as a result of interaction with a faulty neighborhood. Hence it is important to measure the stabilization time as a function of the number of faults in the *start state*, and not as a function of the number of faults in subsequent states. An additional complication is the case of faults which arrive dynamically: a set of faulty processors may cause other, initially correct, processors to become temporarily faulty too. If another set of faults then arrives, the effect would be as if many more processors are faulty. Consequently, we need to measure how long does it take for the system to completely get rid of all the after-effects of a fault, i.e., when is the system fully ready to respond to another batch of faults. We therefore distinguish between *output-stabilization* and *state stabilization*: output stabilization is said to occur once the externally observable portion of the state ceases to change, and state-stabilization is said to occur when the internal state ceases to change as well.

Our results. The main positive result in this paper (Theorem 3.1) is an algorithm for a basic problem called the *persistent bit* problem [10], where the goal is to retain the value of a common replicated bit across the system in spite of transient faults which may corrupt processors state arbitrarily. The bit value is required to be equal at all processors, and the common value should persist across faults so long as in any fault event, a majority of the processors remains correct. Let n denote the number of nodes in the system, and let f denote the number of faulty processors in the start state. The algorithm guarantees that the output is recovered everywhere in $O(f)$ time units, and that complete state stabilization occurs in $O(\mathbf{diam})$ time, for $f < n/2$ (the case of $f \geq n/2$ is not treated in this abstract). The algorithm is fairly robust: f needs not be known in advance, and the network topology may change dynamically. The algorithm can be simplified (and its space and communication requirements are reduced) if an upper bound on f is known *a-priori*. The main drawbacks of the basic algorithm are its high space and communication complexity, and the assumption that execution proceeds in synchronous rounds.

Our main negative result (Theorem 4.3) is that the output stabilization time and the state stabilization time achieved by our algorithm are the best possible in general.

We demonstrate the universality of our approach by presenting an algorithmic transformer which, when given as input a distributed non-reactive, non-stabilizing protocol, produces a time-adaptive version of this protocol, with output stabilization time $O(f)$ and state stabilization time $O(\mathbf{diam})$. This result is formally stated in Theorem 5.1.

Related Work. The study of self-stabilizing protocols was initiated by Dijkstra [5]. Reset-based approaches to self-stabilization are described in [9, 1, 3, 4]. In reset-based stabilization, the state is constantly monitored; if an error is detected, a special reset protocol is invoked, whose effect is to consistently establish a pre-specified global state from which the system can resume normal operation. One of the main drawbacks of this approach is that the detection mechanism triggers a system-wide reset in the face of the slightest inconsistency.

The distinction between output stabilization and state stabilization has been used and discussed in a number of papers. For example, in [2] it is noted that the output stabilizes in $O(\mathbf{diam})$ time, while the state stabilization time may be much larger. Parlati and Yung [12] and Dolev *et al.* [6] study a few cases where state stabilization coincides with output stabilization. Ghosh *et al.* [8] explicitly distinguish between output and state stabilization (called 'fault gap' there).

The papers most closely related to our work are [10, 8]. In [10], the persistent bit problem was introduced, as well as an algorithm with output stabilization time $O(f \log n)$ for $f = O(n/\log n)$, but the number of faults is cumulative: the algorithm cannot correct more than $O(n/\log n)$ faults throughout the execution of the system. In a sense, therefore, the state stabilization time of the algorithm of [10] is infinity. In [8], an algorithm for the following problem is presented: given a self-stabilizing non-reactive protocol, produce another version of that protocol which is self-stabilizing, but whose output stabilization time when $f = 1$ is $O(1)$. The transformed protocol has $O(T \cdot \mathbf{diam})$ state stabilization time, where T is the stabilization time of the original protocol (no analysis is provided for output stabilization time when $f > 1$). The protocol of [8] is asynchronous, and its space overhead is $O(1)$ per link. However, it requires a self-stabilizing protocol to start with, and it may suffer a performance penalty when $f > 1$.

Paper organization. In Section 2 we formalize the model and introduce a few notations. In Section 3 we present our main algorithm for the Persistent Bit problem. In Section 4 we prove lower bounds on stabilization times. In Section 5 we outline the general transformation. In Section 6 we discuss parameterized constructions for known f, and dynamic networks.

2 Model and Notation

The system topology is represented by an undirected graph $G = (V, E)$, where nodes represent processors and edges represent communication links. The number of the nodes is denoted by $n = |V|$. The diameter of the graph is denoted by **diam**. Except for Section 6, in this paper we assume that the network topology is fixed but unknown. For $i \in V$, we define $\mathcal{N}(i) = \{j \mid (i,j) \in E\}$, called the *neighbors* of i. A *distributed protocol* is a specification of the space of *local states* for each node and a description of the *actions* which modify the local states. For simplicity, we abstract the underlying communication mechanism by assuming that actions may depend only on the local state and the state of adjacent nodes. A *global state* is a mapping from the nodes to their local state. An *execution* of the system is a sequence of synchronous steps: at each step (also called *pulse*), each node reads its own variables, as well as the variables of its neighbors, and changes its local state according to the actions specification. Time is measured by the number of steps.

An *input assignment* (respectively, *output assignment*) is a mapping from node names to a given input domain (resp., output range). A *non-reactive problem*, sometimes called an *Input/Output relation*, is a binary relation associating a set of *allowed* output assignments with each possible input assignment. (For example, in the minimum spanning tree problem, the input assignment consists of edge weights, and the output assignment consists of tree edges.) We assume that the state of each node contains designated input and output registers. A problem is said to be *solved* in a given global state if the contents of the input and output registers in that state satisfy the relation specification. A distributed protocol is said to *solve* a problem if when given an input assignment in the designated input registers, it eventually reaches a global state in which the problem is solved.

A *self-stabilizing* protocol has no specification of initial states; instead, it has a *legality predicate* over the global state space, and the requirement is that starting from any state, the system eventually reaches a legal state. For non-reactive protocols, we also require *quiescence*: the successor of each legal state s is s itself. The maximum number of steps to reach a legal state, over all possible start states, is called the *state stabilization time* of the protocol. The *output stabilization time* is the maximum number of steps until the output registers start satisfying the problem specification predicate continuously. We remark that our definition of legal states is in the spirit of [12, 6], and is justified by our separation between output stabilization and state stabilization.

For a given global state s, we define the *fault number* of s to be the minimal number of local states which need to be changed to yield a legal global state. A more operational definition of the fault number involves an adversary, which gets to arbitrarily change the state of f nodes initially in a legal state. Note that the two definitions coincide only when the number of faults according to both is smaller than $n/2$: if a majority of the processors is corrupted, there is no way in general to determine which processors are faulty and which are correct. If the output stabilization time of an algorithm depends only on the fault number then the algorithm is said to be *fault local*.

Typographical Conventions. When discussing a protocol, state variables are represented using teletype font, with a subscript indicating the node in which the variable is located. For example, $\texttt{dist}_i$ denotes the "distance" variable at node i, whose value may be arbitrary. Graph properties are represented using a boldface font, as in $\mathbf{dist}(i,j)$ which denotes the true distance in the graph between nodes i and j.

3 The Persistent Bit Problem

In this section we present our main positive results: upper bounds on the persistent bit problem. We start with a short discussion of the problem, and then present the algorithm.

The problem. The Persistent Bit problem is defined as follows. Each node maintains an externally observable *output bit* which satisfies the following conditions.

- *Eventual Agreement.* All output bits must be equal. If a fault changes the state of some nodes in the system, equality may be reached only after an output stabilization period.
- *Persistence.* Let f denote the number of faulty processors in the start state. If $f < \frac{n}{2}$, then the eventual common value of the output bits is equal to the common value of the majority of the nodes in the start state.

Note that the global state may be faulty even if all output bits are equal: this is because the state may have components other than the output bits.

Our goal is to find a protocol with the smallest possible output and state stabilization time. It is known how to get close to each requirement separately. [10] presents a voting-based protocol which solves the problem in $O(f \log n)$ time for $f = O(\frac{n}{\log n})$, but that protocol is not self-stabilizing, which means (among other things) that the number of faults is counted since system initialization. On the other hand, it is possible to obtain a self-stabilizing algorithm using an efficient reset protocol (e.g., [2])—invoke reset whenever a pair of adjacent nodes disagree on the output bit value. However, global reset results in $\Omega(\mathbf{diam})$ output stabilization time; in addition, it is not immediately clear how to guarantee persistence. In this section, we prove the following result.

Theorem 3.1 *The Persistent Bit problem can be solved by a protocol with output stabilization time $O(f)$ and state stabilization time $O(\mathbf{diam})$.*

(The case of $f \geq \frac{n}{2}$ is not treated in this abstract.) In Section 4, we shall prove that each of the stabilization times above is the best possible. We remark that a simplified (and more efficient) solution is possible if we assume that an upper bound on f is known *a priori* (see Section 6).

3.1 Overview of the protocol

The main difficulty in a time-adaptive solution to the persistent bit problem is that output values are required to change quickly, while old information must not be deleted too early. To see that, consider a tightly fault-local algorithm (an algorithm is tightly fault-local if its output stabilization time is $O(f)$). Suppose that a node i starts with output bit 0, and all its neighbors up to distance t start with output bit 1. Since at the first t steps i must deem itself faulty, i is forced to change its output value to 1 after $O(1)$ steps, and keep it 1 at least through time t. However, it may later turn out that i is correct and all its close neighbors are faulty. Note that if i removes all traces of its previous output value in the first t steps, i becomes, in effect, an additional faulty node, which might affect other non-faulty nodes later.[1] The intuitive conclusion is that nodes should not erase their old state even when they flip the value of their output bit. On the other hand, if the original value is never forgotten, then additional faults have cumulative effect, rendering the solution non-stabilizing. A satisfactory solution to the Persistent Bit problem, which is both fault-local and self-stabilizing, need therefore hit a delicate balance between keeping old information and modifying it.

Our solution consists of two parts: one responsible for speedy stabilization of the output, and the other for prudent state stabilization (see Figure 1). Each node has, in addition to the externally visible 'output bit,' an 'input bit', which serves as a long term memory. In a legal state, all these bits (in all nodes) are equal. The main component of the output stabilization part is the *regulated broadcast* protocol, which ensures that in $O(f)$ time, each node knows the true value of the input bits of sufficiently many nodes. The output bit is computed locally by a simple majority rule. The key to state stabilization is the *input fixing* protocol, which guarantees that all faulty input bits are corrected in $O(\mathbf{diam})$ time units, while making sure that input values at non-faulty nodes never change. Below, we describe each part in detail.

[1] Interestingly, in [11] it is shown that there are graphs where a subset of $\Theta(n^{2/3}\mathbf{diam}^{1/3})$ nodes are the majority in all neighborhoods (up to distance about $\frac{\mathbf{diam}}{2}$) for all nodes in the graph.

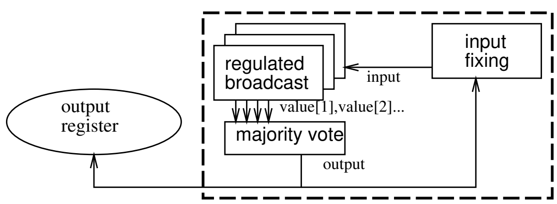

Figure 1: *Schematic structure of the solution: the dashed frame contains the Persistent-Bit protocol.*

3.2 The Regulated Broadcast protocol

The goal of the regulated broadcast protocol, abbreviated RB hereafter, is that each node will have a faithful replica of the input value of every node in the system (including faulty nodes). These values are used to compute the local output bit by a majority rule. For now, we assume that input bits at correct nodes never change. In this case, it is sufficient for fault-locality that in $O(f)$ time, there will be at least $f+1$ correct estimates of non-faulty nodes, and that the only values contradicting non-faulty values may come from faulty nodes.

Note that flooding-based broadcast cannot be used: consider the case where a node i is connected to the network only through a single node j. If j is faulty, a flooding broadcast might result in j corrupting all remote estimates of i's value by broadcasting a wrong value on behalf of i. In general, if flooding is used for broadcast, a single fault at an articulation point can cause a large set of nodes to appear faulty by corrupting their broadcasted value (see Figure 2).

The RB protocol avoids this problem by using the old trick of slowing down to avoid accidents. Specifically, RB builds a tree rooted at each input value, and uses flooding to forward the root value of that tree. However, this 'broadcast wave' is slowed down to half speed (in synchronous systems, this is done by

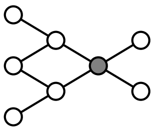

Figure 2: *A single faulty node (gray) may hide many non-faulty values.*

taking a step every second pulse). At the same time, nodes keep verifying the integrity of the tree and the broadcasted information; if an inconsistency is found, a 'reset wave' is initiated; this wave progresses at full speed down the broadcast tree, erasing all estimates and tree structure as it goes. Because of the speed difference, a wrong value cannot reach too far before it is eliminated.

Constants

V	:	the set of nodes
$\mathcal{D}$	:	an upper bound on **diam**
$\mathcal{N}(i)$	:	the set of neighbors of i

State for Node i

$\mathtt{value}_i[V]$	:	array of $\{0, 1, \perp\}$
$\mathtt{parent}_i[V]$	:	array of $\mathcal{N}(i) \cup \{\mathtt{null}\}$
$\mathtt{dist}_i[V]$	:	array of $\{1, \ldots, \mathcal{D}\} \cup \{\infty\}$
$\mathtt{output}_i$		$\in \{0, 1\}$

Shorthand

$$\begin{aligned}
&\mathit{inconsistent}_i(j) \equiv \\
&\Big((i = j) \Rightarrow (\mathtt{parent}_i[j] \neq \mathtt{null}) \vee (\mathtt{dist}_i[i] \neq 0)\Big) \bigwedge \\
&\Big((i \neq j) \Rightarrow (\mathtt{value}_i[j] \neq \mathtt{value}_{\mathtt{parent}_i[j]}[j]) \vee \\
&\qquad (\mathtt{dist}_i[j] \neq \mathtt{dist}_{\mathtt{parent}_i[j]}[j] + 1) \vee \\
&\qquad (\mathtt{dist}_i[j] \neq \min\{\mathtt{dist}_k[j] + 1 \mid k \in \mathcal{N}(i)\}) \vee \\
&\qquad (\mathtt{value}_{\mathtt{parent}_i[j]}[j] = \perp)\Big)
\end{aligned}$$

Actions for Node i

at every pulse:

$\mathtt{parent}_i[i] \leftarrow \mathtt{null}$; $\mathtt{dist}_i[i] \leftarrow 0$

for each $j \in V$ **do**

  if $(i \neq j) \wedge \mathit{inconsistent}_i(j)$ **then**

    $\mathtt{parent}_i[j] \leftarrow \mathtt{null}$

    $\mathtt{dist}_i[j] \leftarrow \infty$

    $\mathtt{value}_i[j] \leftarrow \perp$

at every second pulse:

for each $j \in V$, $j \neq i$ **do**

  $\mathtt{parent}_i[j] \leftarrow k$ such that $\mathtt{dist}_k[j] = \min\{\mathtt{dist}_l[j] \mid l \in \mathcal{N}(i)\}$

  $\mathtt{dist}_i[j] \leftarrow \mathtt{dist}_{\mathtt{parent}_i[j]} + 1$

  $\mathtt{value}_i[j] \leftarrow \mathtt{value}_{\mathtt{parent}_i[j]}[j]$

$\mathtt{output}_i \leftarrow$ majority $\{\mathtt{value}_i[j] \mid j \in V\}$

Figure 3: *Code for output stabilization at processor i.*

Pseudo-code for output stabilization is presented in Figure 3. The $\mathtt{value}_i[j]$ entry is the current estimate of node i for the input bit of node j ($\mathtt{value}_i[i]$ is the input bit for node i). The underlying broadcast protocol is based on the standard Bellman-Ford technique. The majority function ignores $\perp$ values and outputs 0 in case of a tie.

We now analyze the RB protocol. We consider a representative tree rooted at a non-faulty node j, since the different trees operate independently (we ignore trees rooted at faulty nodes: they can have any value). To facilitate the discussion, we introduce a graph theoretic concept, and a few terms about executions of the algorithm.

Definition 3.1 *Let $i \in V$. The* depth *of i is* $\mathbf{depth}(i) \stackrel{\text{def}}{=} \max\{\mathbf{dist}(i, j) \mid j \in V\}$.

Definition 3.2 *Let $j \in V$, and fix a global state.*

(1) A node i is correct with respect to $j \neq i$ *if all the following conditions hold true.*

- $\mathtt{value}_i[j] = \mathtt{value}_j[j]$
- $\mathtt{dist}_i[j] = \mathbf{dist}(i, j)$
- $\mathtt{dist}_i[j] = \min\{\mathtt{dist}_k[j] + 1 \mid (k, i) \in E\}$
- $\mathtt{dist}_i[j] = \mathtt{dist}_{\mathtt{parent}_i[j]}[j] + 1$

(2) A node i is correct with respect to itself *if* $\mathtt{dist}_i[i] = 0$ *and* $\mathtt{parent}_i[i] = \mathtt{null}$.

(3) $\mathtt{parent}_i[j]^t$ *is the t-th ancestor of node i in j's tree obtained inductively by following t* $\mathtt{parent}[j]$ *pointers starting at i. By convention,* $\mathtt{parent}_i[j]^0 \stackrel{\text{def}}{=} i$.

The following lemma is a minor variant of the standard self-stabilization property of the Bellman-Ford algorithm.

Lemma 3.2 *Let $t \geq 0$. If* $\mathtt{value}_j[j]$ *does not change in a time interval $[0, 2t + 1]$, then for all nodes i with* $\mathbf{dist}(i, j) \leq t$, *$i$ is correct with respect to j at time* $2t + 1$.

By Lemma 3.2, faithful estimates of input values which do not change are established quickly. We now show that unfaithful estimates of input values disappear quickly too. This is done by means of local detection: the local predicate $\mathit{inconsistent}_k(j)$, given in Figure 3, guarantees that if the tree is corrupted, then some node will be able to detect it, as shown in the next lemma.

Lemma 3.3 *Let $i \in V$, let $j \in v$ be non-faulty, and suppose that* $\mathtt{value}_i[j] \neq \mathtt{value}_j[j]$. *Then for some* $0 \leq t \leq f$, *there exists a node* $k = \mathtt{parent}_i[j]^t$ *such that* $\mathit{inconsistent}_k(j)$ *holds.*

Proof: By contradiction. Define P to be the set of nodes $\{\mathtt{parent}_i[j]^t \mid 0 \leq t \leq f\}$. Suppose that for all $k \in P$ we have that $\mathit{inconsistent}_k(j)$ is false. By definition and transitivity, $\mathtt{value}_i[j] = \mathtt{value}_k[j]$ for all $k \in P$. We consider two cases: if $|P| = f+1$ or $j \in P$, then there exists a non-faulty $k \in P$. For that k we have $\texttt{value}_k[j] = \texttt{value}_j[j]$, a contradiction to the assumption that $\texttt{value}_i[j] \neq \texttt{value}_j[j]$. Otherwise, $|P| \leq f$ and $j \notin P$. In this case, for some $k \in P$ (the last one in the sequence), we have $\texttt{parent}_k[j] = \texttt{null}$ and $j \neq k$, and hence $\mathit{inconsistent}_k(j)$ is true, a contradiction. ▮

The following lemma proves the basic property of this correction wave: all wrong estimates disappear quickly.

Lemma 3.4 *Let $j \in V$ be non-faulty, and assume that $\texttt{value}_j[j]$ does not change for $t \geq 2 \cdot \min(\mathbf{depth}(i), f)$ time units. Then at time t we have that for all $i \in V$, $\texttt{value}_i[j] \in \{\texttt{value}_j[j], \perp\}$.*

Proof: For $t \geq 2 \cdot \mathbf{depth}(i)$, the claim holds by Lemma 3.2. So assume that $f < \mathbf{depth}(i)$. Given a state s, let $\delta(s)$ denote the maximal length of a parent chain in state s that starts with a node which is incorrect w.r.t. j, and ends with the first inconsistent ancestor. Formally, in state s, let $t_i(s) = \min\left\{t \mid \mathit{inconsistent}_{\texttt{parent}_i[j]^t}(j)\right\}$ ($t_i(s)$ may be undefined if i is correct w.r.t. j), and define $\delta(s) = \max\{t_i(s) \mid t_i(s) \text{ is defined}\}$, and $\delta(s) = -1$ if $t_i(s)$ is undefined for all $i \in V$. By Lemma 3.3, we have that in the start state s_0, $\delta(s_0) \leq f$. By definition, $\delta(s) \geq 0$ if and only if there is some node which is incorrect w.r.t. j. To prove the lemma, we argue that δ is decremented by at least one unit every two steps. Let s be a given state, and let s' denote the state of the system after two steps. Let i be a node such that i is incorrect w.r.t. j and $\mathit{inconsistent}_{\texttt{parent}_i[j]^{\delta(s)}}(j)$ is true. Then by the code, in the next two steps, we will have $\mathit{inconsistent}_{\texttt{parent}_i[j]^{\delta(s)-2}}(j)$ since the reset wave advances at the rate of one link per pulse. However, only nodes which are at distance 1 from i can become incorrect w.r.t. j, since the broadcast wave takes only one step. ▮

We can now prove that the output stabilizes quickly, provided that the non-faulty input bits remain fixed.

Theorem 3.5 *If the input values at all non-faulty nodes do not change and $f < n/2$, then after $\min(2 \cdot \mathbf{diam}, 4f + 2)$ steps all output values are equal to the input values of non-faulty nodes.*

Proof: Fix a node $i \in V$, and suppose without loss of generality that $\texttt{value}_j[j] = 0$ for all non-faulty nodes j. If $4f + 2 \geq 2 \cdot \mathbf{diam}$ the claim follows from Lemma 3.2. So suppose now that $4f + 2 < 2 \cdot \mathbf{diam}$. By Lemma 3.2, after $t \geq 2 \cdot (2f + 1)$ time units, $\texttt{value}_i[j] = 0$ for at least $f+1$ nodes j, since there are at least $f + 1$ non-faulty nodes whose distance from i is no more than $2f + 1$. Moreover, by Lemma 3.4, after $t \geq 2f$ time units, $\texttt{value}_i[j] \neq 1$ for all non-faulty nodes j. It therefore follows from the majority rule used by the algorithm that after $4f + 2$ time units $\texttt{output}_i = 0$ for all nodes i, as required. ▮

We remark that the output stabilization time of the regulated broadcast algorithm is better than the output stabilization time of the mending algorithm of [10], but the algorithm of [10] guarantees complete *quiescence* (which is stronger than output stabilization) after $O(f \log n)$ time steps.

Observe that the proof of Theorem 3.5 actually shows that the output of each node i stabilizes in at most $1 + 2 \cdot \min(\mathbf{depth}(i), 2f + 1)$ time units. This fact is key to the input fixing algorithm, presented next.

3.3 The Input Fixing protocol

One of the crucial assumptions the regulated broadcast protocol relies on is that input bits of non-faulty nodes never change. However, if the algorithm never changes the value of input bits, the result would be that the effect of faults is accumulated, since once an input bit is compromised, its corrupted value would linger and potentially prevent output stabilization for additional faults. The goal of the input fixing protocol is to change the input bits as quickly as possible. The key idea in the solution is that if the output value does not change for a sufficiently long time, then it is safe to assume that it is correct.

In more detail, the idea is as follows. It follows from Lemma 3.4 and Theorem 3.5 that the output value at node i is correct if it remains unchanged for $T(i, f) \stackrel{\text{def}}{=} 1+2\cdot\min(\mathbf{depth}(i), 2f+1)$ time units. The problem can therefore be reduced to (1) constructing a self-stabilizing timer, and (2) estimating the value of $T(i, f)$: if we have both, we can set the timer to count steps, reset it to 0 every time the output value changes, and change the input value when the timer reaches $T(i, f)$. However, both problems seem to require some work in the presence of transient faults which may corrupt the state: (1) it is impossible to tell whether a timer shows the correct count of pulses or is it just a meaningless value assigned to it by a fault; and (2) it not immediately clear how can one estimate $T(i, f)$.

We first note that constructing a timer is not really a problem: no harm is done if timers *at faulty nodes* expire prematurely, since the worst possible consequence is that the faulty node changes its input prematurely. This is not a problem, since the proof of Theorem 3.5 does not require the input bits of faulty nodes not to change. Thus, it remains to construct a self stabilizing timer which does not expire prematurely at non-faulty nodes. This can be done simply by counting the steps. Computing the value of $T(i,f)$ in which the timer expires seems harder, since it may require dependence on faulty nodes.

We solve this difficulty by upper-bounding $T(i,f)$ with a little self-stabilizing protocol which computes $\mathbf{depth}(i)$, which preserve a certain invariant even before stabilization.

```
State for Node i
    parent_i[V] :  array of N(i) ∪ {null}
    dist_i[V]   :  array of {1,...,D} ∪ {∞}
    depth_i     ∈  {0,...,D} ∪ {∞}
    count_i     ∈  {0...2·D+2}
    output_i    ∈  {0,1}        (from Fig. 3)
    input_i     ∈  {0,1}        (called value_i[i] in Fig. 3)

Actions for Node i
at every pulse:
  parent_i[i] ← null; dist_i[i] ← 0
  for all j ∈ V, j ≠ i do
      parent_i[j] ← k such that
                  dist_k[j] = min{dist_l[j] | l ∈ N(i)}
      dist_i[j] ← dist_{parent_i[j]} + 1
  depth_i ← max{dist_i[j]}
  if depth_i or output_i are modified then count_i ← 0
  if count_i ≥ 2·(depth_i + 1) then
          input_i ← output_i
          count_i ← 2·D + 2
  else count ← (count + 1) mod (2·D + 3)
```

Figure 4: *Input fixing protocol: code for processor i.*

Pseudo-code for the input fixing protocol is presented in Figure 4. Informally, the algorithm constructs a Bellman-Ford tree rooted at each node, and obtains a bound on the depth of the node simply by taking a maximum over all distance estimates the node currently has. (For simplicity, we assume here that these trees are distinct from the trees constructed by the RB protocol.) Clearly, this algorithm stabilizes in $O(\mathbf{diam})$ time units. The new twist in this algorithm is using the counter. At a legal state, the counter is fixed at its highest value. When the depth estimate or the output value change, the counter is reset to 0. (Recall that the $\mathtt{output}_i$ variable is maintained by the output stabilization protocol.) The counter is incremented by one at each subsequent step, until its value is greater than twice $\mathtt{depth}_i$. As we show, at that point, either $\mathtt{depth}_i \geq \mathbf{depth}(i)$ or i is faulty. In any case, i can safely assign $\mathtt{output}_i$ to be its input bit $\mathtt{value}_i[i]$.

We now analyze the input fixing protocol more formally. We first state an immediate property of the Bellman-Ford algorithm. The importance of this property is that it holds before stabilization.

Lemma 3.6 *Let $i,j \in V$, and let $t \geq 0$. Then after taking t steps starting from any state,* $\mathtt{dist}_i[j] \geq \min(t, \mathbf{dist}(i,j))$.

The following corollary shows that the fact that the lower bound of Lemma 3.6 holds even before stabilization prevents premature assignment of input values by the input fixing protocol.

Corollary 3.7 *Suppose that for some node $i \in V$, we have that $\mathtt{depth}_i$ remains unchanged for $\mathtt{depth}_i + 1$ time units. Then $\mathtt{depth}_i \geq \mathbf{depth}(i)$.*

Proof: Let $d = \mathtt{depth}_i$, and suppose that $\mathtt{depth}_i$ remains fixed for $d+1$ time units. Suppose, for contradiction, that $d < \mathbf{depth}(i)$. Then by definition, for some $j \in V$, $\mathbf{dist}(i,j) \geq d+1$. By Lemma 3.6, after $d+1$ steps, $\mathtt{dist}_i[j] \geq d+1$, which is a contradiction, since by the code, $\mathtt{depth}_i \geq \mathtt{dist}_i[j]$ for all $j \in V$. ∎

From Corollary 3.7, Lemma 3.2, and Theorem 3.5, we can conclude the following.

Theorem 3.8 *Suppose that in the start state there are $f < n/2$ faulty nodes.*

(1) Input values at non-faulty nodes never change.

(2) Input values at all nodes are correct after $O(\mathbf{diam})$ steps.

(3) The state of the input fixing protocol stabilizes in $O(\mathbf{diam})$ steps.

Proof: (1) Suppose, for the sake of contradiction, that some non-faulty node changes its input value during the execution. Let j be the first such node. By the code (Figure 4), the following must hold true when j changes its input value:

- $\mathtt{count}_j = 2 \cdot \mathtt{depth}_j + 1$, and
- $\mathtt{value}_j[j] \neq \mathtt{output}_j = \mathrm{majority}(\mathtt{value}_j)$.

Since j is correct, in the start state we have $\mathtt{value}_j[j] = \mathtt{output}_j$, and hence at some point during the execution j changed its output value. At that step, the code dictates that $\mathtt{count}_j$ was reset to 0. Therefore, since $\mathtt{count}_j$ is incremented at each step, we must have that $\mathtt{depth}_j$ and $\mathtt{output}_j$ did not change their values for at least $2 \cdot \mathtt{depth}_j + 1$ time units. It follows from Corollary 3.7 that $\mathbf{depth}(j) \leq$

$\mathtt{depth}_j$, and hence at least $2 \cdot \mathbf{depth}(j)+1$ time units have elapsed. By our assumption that j is the first correct node to change its input value (and thus input values in all non-faulty nodes have not changed so far) we can apply Theorem 3.5 to conclude that at that time, the value of $\mathtt{output}_j$ must be correct at this step, a contradiction.

(2) Let j be a faulty node. By Theorem 3.5 and by (1) we have that after at most $2 \cdot \mathbf{depth}(j)+1$ time units, $\mathtt{output}_j$ is correct. By Lemma 3.2 and the code, after at most $2 \cdot \mathbf{depth}(j)$ additional time units, the input value will be changed to the correct value. The statement follows from the fact that for all $j \in V$, $\mathbf{depth}(j) \leq \mathbf{diam}$.

(3) By (1) and (2), the input values stabilize after $O(\mathbf{diam})$ time units. The result follows from the standard stabilization argument for the Bellman-Ford algorithm. ∎

4 Lower Bounds on Stabilization Times

In this section we prove that the output and state stabilization times attained by our algorithm for the persistent bit protocol are asymptotically optimal. Both proofs use indistinguishibility arguments; Lemma 4.1 is simple, and Lemma 4.2 is a little more subtle. Both proofs hold even if the network topology is known.

Below, we consider a system whose underlying graph is a line of n nodes. We use the following additional notation. For an integer i, let $L(i)$ denote the leftmost i nodes of the graph; similarly, let $R(i)$ denote the set of the rightmost i nodes. For a global state s and a subset $A \subseteq V$ of nodes, let $s|_A$ denote the projection of s on the nodes of A. We use two legal global states, s_0 with output value 0, and s_1 with output value 1.

We first bound the output stabilization time.

Lemma 4.1 *Let f be the number of faults in the initial state, and assume that $f < n/2$. There is no protocol for the persistent bit problem with output stabilization time less than f, even if f is known a-priori.*

Proof: Fix $f < n/2$. Let s be the global state such that $s|_{L(f)} = s_0|_{L(f)}$ and $s|_{R(n-f)} = s_1|_{R(n-f)}$, and let s' be the global state such that $s'|_{R(f)} = s_1|_{R(f)}$ and $s'|_{L(n-f)} = s_0|_{L(n-f)}$. Intuitively, s is the state of the system which, after being in legal state s_1, was hit by a fault which corrupted the nodes in $L(f)$, and s' is the state of the system which was in legal state s_0 and then a fault changed the state of the nodes in $R(f)$. Suppose, for contradiction, that starting from s, the output of the system stabilizes in time $t < f$. Consider the leftmost node: it cannot distinguish between s and s' in less than f time units, and therefore it must behave identically under both initial states in the first t time units. But by the problem specification, its output in s after t time units should be 0, and its output in s' after t time units should be 1, contradiction. ∎

The lower bound for state stabilization uses the assumptions that f is unknown, and that state do not change after stabilization.

Lemma 4.2 *There is no protocol for the persistent bit problem with state stabilization time $o(\mathbf{diam})$.*

Proof: Let p be a parameter, and define three subsets of the nodes: $A = L(p)$, $C = R(\frac{n-p}{2})$ and $B = V \setminus (A \cup C)$ (see Figure 5). We have that $|A| = p$, and $|B| = |C| = \frac{n-p}{2}$. We prove the lemma by considering a start state with p faulty nodes, and showing that the state stabilization time in this case is at least $\frac{n-p}{4}$. To do that, we consider three scenarios as follows.

In Scenario I, the start states of the nodes are as follows. The nodes in A start with $s_0|_A$, the nodes in B start with $s_1|_B$, and the nodes in C start with $s_1|_C$. In this case the nodes in A are faulty: intuitively, the system was in the legal state s_1, and the nodes in A were hit by a fault which changed their state to $s_0|_A$. Let the the state stabilization time of this start state be t. We will show that $t \geq \frac{n-|A|}{4}$. Let s_2 denote the global state in Scenario I after t steps. We note a few facts about s_2:

(1) s_2 is a legal state with output value 1.

(2) $s_2|_{R(n-|A|-t)} = s_1|_{R(n-|A|-t)}$, i.e., the local states of all nodes, except A and the leftmost t nodes of B, is exactly as in s_1.

Scenario II, intuitively, starts at s_2 after a fault changes the state of the nodes in C to be as in s_0. Formally, the start state of this scenario is s_3, defined by $s_3|_{A\cup B} = s_2|_{A\cup B}$ and $s_3|_C = s_0|_C$. The problem specification requires that the outcome for this scenario is 1. Let s_4 be the state of the system after t steps in this scenario. We note an additional fact which follows from fact (1):

(3) s_4 must lead to a state where the output value is 1 (even though s_4 might not be legal).

Next, consider Scenario III in which the start states are as follows. The nodes in A start with $s_0|_A$, the nodes in B start with $s_1|_B$, and the nodes in C start with $s_0|_C$. In this case the nodes in B are considered faulty: intuitively, the system was in the legal state s_0, and the nodes in B were hit by a fault which changed their state to be $s_1|_B$. The problem specification requires that the outcome for this scenario is 0.

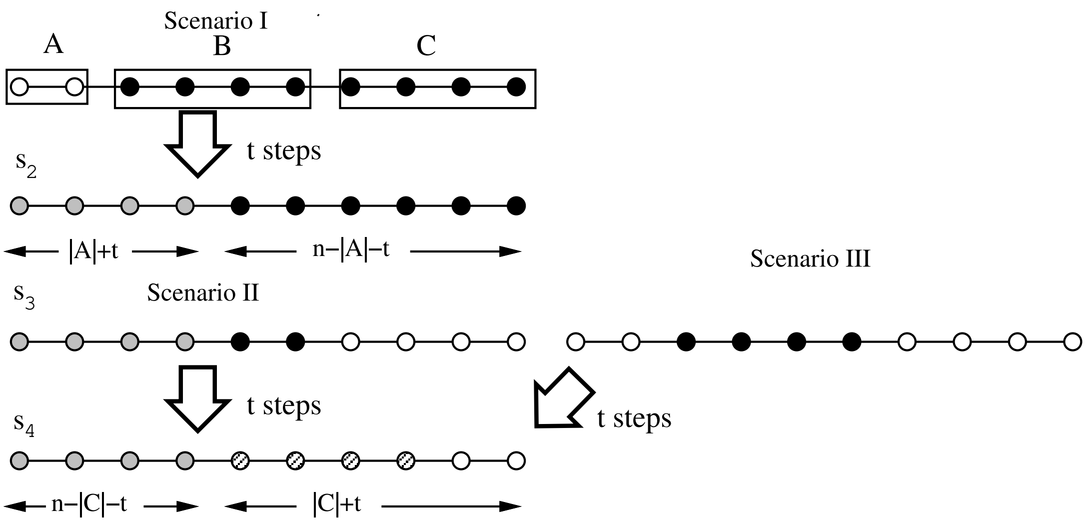

Figure 5: *Example considered in the proof of Lemma 4.2. White nodes are in local state s_0, black nodes are in local state s_1, and gray nodes are in other different states.*

Let s'_4 be the state of Scenario III after t steps. To complete the proof, we claim that if $t \leq \frac{n-|A|}{4}$, then $s_4 = s'_4$, which is a contradiction, since by fact (3), s_4 leads to output 1, while s'_4 must lead to output 0. We now prove that $s_4 = s'_4$. First, note that $s_4|_{L(n-|C|-t)} = s_2|_{L(n-|C|-t)}$ by the quiescence of s_3; also, $s_2|_{L(n-|C|-t)} = s'_4|_{L(n-|C|-t)}$ because the nodes in $L(n-|C|-t)$ take the same first t steps in Scenarios III and I. It follows that $s_4|_{L(n-|C|-t)} = s'_4|_{L(n-|C|-t)}$. Next, note that $s_4|_{R(n-|A|-t)} = s'_4|_{R(n-|A|-t)}$ because the nodes in $R(n-|A|-t)$ take the same first t steps in Scenarios II and III. But if $t \leq \frac{n-|A|}{4}$ then $(n - |C| - t) + (n - |A| - t) \geq n$ which implies that $R(n - |A| - t) \cup L(n - |C| - t)$ is the set of all nodes, and hence $s_4 = s'_4$, as required. ∎

We summarize in the following theorem.

Theorem 4.3 *There is no protocol for the persistent bit problem with output stabilization time $o(f)$ or state stabilization time $o(\mathbf{diam})$.*

5 The General Transformer

In this section we show how to transform a given algorithm for a general non-reactive problem (e.g., Minimum Spanning Tree) to a fault-local self stabilizing one. Our technique is essentially brute force. The result should therefore be viewed just as a possibility result, since the space overhead in our particular method is high.

Let us state the result in the most general (and inefficient) way. Intuitively, the statement below views the state before the output is computed as faulty, and computing the output is viewed as recovering from faults. We do not think that this alternative is an efficient approach. Nevertheless, it unifies elegantly the treatment of computation and fault recovery.

Theorem 5.1 *Let $\mathcal{P}$ be a non-stabilizing protocol solving problem Π. Then there exists a self-stabilizing protocol $\mathcal{P}'$ which solves Π after any number $f < n/2$ faults in the start state with output stabilization time $O(f)$ and state stabilization time $O(\mathbf{diam})$, even if f is unknown.*

Proof Sketch: We outline the transformed protocol $\mathcal{P}'$. The idea is that in $\mathcal{P}'$, each input bit is replicated over all nodes in the system. These bits are maintained by the persistent-bit algorithm. By Theorem 3.5, all the replicated inputs will have the correct values in $O(f)$ time units. Note that each node has all inputs locally available, and it can therefore simulate $\mathcal{P}$ locally in a single time step. It follows that the output values for Π will stabilize in $O(f)$ time units. The state of $\mathcal{P}'$ stabilizes in $O(\mathbf{diam})$ steps by Theorem 3.8. ∎

6 Extensions

Space complexity. One way to reduce the space complexity of our algorithm is to assume that there is a known upper bound F on the number of processors corrupted in a single transient fault. Under this assumption, our algorithm can be converted to an algorithm with output stabilization time $O(f)$, state-stabilization time $O(F)$, and space complexity $O(F)$. The idea is as follows. The output stabilization protocol will use regulated broadcast such that each node has `value` array with only $2F+1$ entries: this is sufficient to in order to compute the correct result when the number of faults is at most F. Similarly, the input fixing protocol needs only to implement a timer which counts up to $4F$, and there is no need to compute the depth of the node.

Dynamic networks. In dynamic networks, links can crash and recover arbitrarily, and crashes and recoveries are detected by the incident nodes. It is straightforward to verify that in our algorithm the output bits will not be affected by such changes. However, topological changes violate the integrity of the underlying tree structures, which may take $O(\mathbf{diam})$ time to regain stability. Fortunately, one can prove that this disturbance is really not damaging to our protocol: if, while recovering from a topological change, f nodes become faulty, the output will still stabilize in $O(f)$ time units. This property is due to the fact that even if the topology changes there are still $f+1$ non-faulty nodes at distance at most $2f+1$ from every node.

Acknowledgement

We thank Oded Goldriech for many very useful comments.

References

[1] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, Sept. 1990. Springer-Verlag (LNCS 486). To appear in *Theoretical Comp. Sci.*

[2] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, California*, pages 652–661, May 1993. Also appeared as IBM Research Report RC-19149(83418).

[3] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 268–277, Oct. 1991.

[4] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset. In *Proc. 8th International Workshop on Distributed Algorithms*, pages 326–339. Springer-Verlag (LNCS 857), 1994.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17(11):643–644, November 1974.

[6] S. Dolev, M. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[7] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *Proc. of the Second Workshop on Self-Stabilizing Systems*, pages 3.1–3.15, May 1995.

[8] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemamraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, May 1996.

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

[10] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 1995.

[11] N. Linial, D. Peleg, Y. Rabinovich, and M. Saks. Sphere packing and local majorities in graphs. In *Proceedings of the 2nd Israel Symposium on Theory of Computing and Systems*, pages 141–149, June 1993.

[12] G. Parlati and M. Yung. Non-exploratory self-stabilization for constant-space symmetry-breaking. In J. van Leeuwen, editor, *Proceedings of the 2nd Annual European Symposium on Algorithms*, pages 26–28, Sept. 1994. LNCS 855, Springer Verlag.