Stabilizing Time-Adaptive Protocols*

Shay Kutten
Dept. of Industrial Engineering & Management
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

February 8, 1998

Abstract

We study the scenario where a transient batch of faults hit a minority of the nodes in a distributed system by corrupting their state. We concentrate on the basic *persistent bit* problem, where the system is required to maintain a 0/1 value in the face of transient failures by means of replication. We give an algorithm to stabilize the value to a correct state quickly; that is, denoting the unknown number of faulty nodes by f, our algorithm recovers the value of the bit at all nodes in $O(f)$ time units for any $f < n/2$, where n is the number of all nodes. Moreover, complete state quiescence occurs in $O(\mathbf{diam})$ time units, where **diam** denotes the actual 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. (Strict self-stabilization requires recovery for $f \geq n/2$ as well.) We prove matching lower bounds on both the output stabilization time and the state quiescence time. Using our persistent bit algorithm, we present a transformer which takes a distributed non-reactive non-stabilizing protocol $\mathcal{P}$, and produces a protocol $\mathcal{P}'$ which solves the problem $\mathcal{P}$ solves, with the additional property that if a batch of faults changes the state of $f < n/2$ of the nodes, then the output is recovered in $O(f)$ time units, and the state stabilizes in $O(\mathbf{diam})$ time units. Our upper and lower bounds are all proved in the synchronous network model.

1 Introduction

Stabilizing distributed systems recover from a particularly devastating type of fault: a state-corrupting fault. During a state corrupting fault, the bits of the volatile memory in an affected

*A preliminary version of this paper appeared in *Proc. ACM Symposium on Principles of Distributed Computing,* Aug. 1997.

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

processor may be arbitrarily flipped. Their resilience against such faults make stabilizing systems highly desirable, as it generalizes resilience against any kind of transient fault. Naturally, stabilization is quite expensive in terms of computational resources. For example, one of the common methods to make a system stabilizing is the *reset* paradigm, which offers 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 solution we call *time-adaptive protocols*: these are protocols which recover from a limited number of state-corrupting 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.

We look at one of the basic building blocks of time-adaptive protocols, called the *persistent bit* problem [19]. In this problem, the task is to maintain the value of a bit in spite of state-corrupting faults. In some sense, the persistent bit problem captures the essence of the state-corrupting faults: maintaining a value is a trivial task if no faults are involved, but it is not a simple matter in face of state corruption. Clearly, solving this problem alone does not provide a full answer to time-adaptivity, since it deals with static values. However, it seems that any interesting time-adaptive protocol embeds a solution to the persistent bit problem in it. The study of time-adaptive protocols with dynamically changing values is left for further research.

Before we proceed to state our results, we make a slight technical digression to explain an important subtle point implicit in the notion of time adaptivity. Loosely speaking, a protocol is called time-adaptive if its recovery time depends only on the number of state-corrupted nodes (rather than, say, the total number of nodes). In this paper, a faulty processor is defined (see definition below) to be one whose state differs from its state in a given (unknown, legal) state. Note that it is possible for a node which executes the protocol to be considered initially non-faulty and later faulty, if its neighborhood is faulty. Hence it is important to measure the stabilization time as a function of the number of faults in the *start state* (immediately following the faults), and not as a function of the number of faults in subsequent states. This phenomenon is exposed in the case of additional faults, as demonstrated by the following scenario. Initially, f_1 processors are faulty. After a few steps, it may be the case that x processors are called faulty by our definition, for some $x > f_1$. If f_2 additional processors are then hit by another batch of faults, the effect is as if there are now $x + f_2 > f_1 + f_2$ faulty processors, even though only $f_1 + f_2$ processors were directly affected by a fault. The important conclusion is that for time adaptivity, one needs to measure how long it takes for the system to completely get rid of all the after-effects of the faults, i.e., when is the system fully ready to respond to another batch of faults. We therefore distinguish between the concepts of *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. As mentioned above, the results in this paper concern the *persistent bit* problem [19], 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, so long as the state at a majority of the processors is not faulty. The bit value is required to be equal at all processors, and the common value should persist across faults. Our main positive result (Theorem 3.1) is an algorithm for the persistent bit problem. Let n denote the number of nodes in the system (we use the terms "nodes" and "processors" interchangeably), and let f denote the number of faulty processors in the start state. Let **diam** be the actual diameter of the network. The algorithm guarantees, for any $f < n/2$, that the output is recovered everywhere in $O(f)$ time units, and that complete state stabilization occurs in $O(\mathbf{diam})$ time. The algorithm is fairly robust in the sense that f and the network topology need not be known in advance. The algorithm can be simplified (and its space and communication requirements are reduced) if a smaller upper bound on f is known *a priori*. However, our result falls short of the original goal in several respects. First, strictly speaking, the algorithm is not *self*-stabilizing, where the requirement is to recover from any number of faults $f \leq n$ (note, however, that preserving the value of a bit is somewhat meaningless when $f \geq n/2$); secondly, the algorithm is stated and proved correct only for the case of executions which proceed in synchronous rounds; and thirdly, the algorithm has high space and communication complexity.

Our negative results (Theorem 4.3) give lower bounds on the stabilization complexity of the persistent bit problem. We show that our algorithm is optimal in terms of stabilization times: there is no algorithm for the persistent bit problem with output stabilization time $o(f)$, or state stabilization time $o(\mathbf{diam})$. We note that the lower bounds are proved for the synchronous executions model (and hence for the asynchronous model *a fortiori*).

We proceed to 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 $O(\mathbf{diam})$, for $f < n/2$, under the assumption that the inputs are replicated over all nodes. This result is formally stated in Theorem 5.1.

Related Work. The study of self-stabilizing protocols was initiated by Dijkstra [12]. *Reset-based* approaches to self-stabilization are described in [18, 5, 8, 9, 15]. 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 correct 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 [7] it is noted that the output stabilizes in $O(\mathbf{diam})$ time, while the state stabilization time may be much larger. Parlati and Yung [23] and Dolev *et al.* [14]

study a few cases where state stabilization coincides with output stabilization. Ghosh *et al.* [17] explicitly distinguish between output and state stabilization (called 'fault gap' there). In [15], a distinction is made between a state-corrupting fault which triggers a reset, and a topological change which results in a milder effect.

The papers most closely related to our work are [19, 17, 3]. In [19], 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 certain sense, therefore, the state stabilization time of the algorithm of [19] is infinity. In [17], 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 is $O(1)$ if $f = 1$. 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 [17] 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 in the case of $f > 1$. In [3] faults are stochastic, and consequently the correctness of information can be decided with any desired certainty less than 1. Under this assumption, a time-adaptive algorithm is presented. The algorithm handles both Input-Output relations, and reactive tasks. Additional examples for the special case of $O(1)$ recovery time appear in [11, 22, 15, 21].

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.

2 Model and Notations

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 $\mathbf{diam}$. We shall assume that there is a known upper bound on the diameter of the network, denoted by $\mathcal{D}$. This upper bound serves only for the purpose of having finite space protocols. For $i \in V$, we define $\mathcal{N}(i) = \{j \mid (i,j) \in E\}$, called the *neighbors* of i. For the purpose of algorithms, we do not assume that the set of edges in the network is known in advance, i.e., algorithms are required to work on any topology. 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 and the variables of its neighbors, and then 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. 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.

We assume that each protocol has a *legality predicate* over the global state space. 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 perturbed to yield a legal global state (see, e.g., [15]). A more intuitive definition of the fault number (see [19]), involves an adversary which gets to arbitrarily change the state of f node in a legal state, and then the operation of the protocol is resumed. Formally, this definition requires two global states: a start state s_0 and a 'reference state' s_{-1}, where s_{-1} is legal; a processor is called faulty if its local state differs in s_0 and s_{-1}. A protocol is called *f-stabilizing* (or *stabilizing* for short) if starting from a state with fault number at most f, it eventually reaches a legal state and remains in a legal state thereafter. A protocol is called *self-stabilizing* if it is f-stabilizing for $f = n$.

For non-reactive protocols, we require *quiescence* in addition to stabilization: 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 stop changing. We remark that our definition of legal states is in the spirit of 'non-exploratory protocols' [23] and 'silent stabilization' [14], and is justified by our separation between output stabilization and state stabilization.

If the output stabilization time of an algorithm depends only on the fault number then the algorithm is said to be *fault local*, or *time-adaptive.*[1].

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.

[1] The term "time adaptive" was used before in different contexts, for example see [6]

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 (this is a degenerate case of input/output relation: there is no input).

- *Eventual Agreement.* All output bits must be equal, except perhaps for a finite time, called *output stabilization time* immediately following a batch of fault.
- *Persistence.* If the number of faults f in a given start state satisfies $f < n/2$, 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 a global state may be faulty even if all output bits are equal: this is because in general, states 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. [19] 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 stabilizing in the sense that the state is never completely recovered, and hence 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., [7])—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 *There exists a protocol for the persistent bit problem such that if the local states of $f < n/2$ of the nodes are changed arbitrarily, then the output bits are restored everywhere in $O(f)$ time units, and complete state stabilization occurs in $O(\mathbf{diam})$ time units, where* **diam** *denotes the diameter of the network.*

In Section 4, we shall prove that output- and state-stabilization times above is the best possible. In Section 6 we discuss a simplified (and more efficient) solution under the assumption that a better upper bound on f is known. (The case of $f \geq \frac{n}{2}$ is not treated in this paper.)

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 given network, a node i_0 in it, and let N_0 be the set of all nodes at distance t or less from i_0. Suppose that node i_0 starts with output bit 0, and all the nodes in $N_0 - \{i\}$ start with output bit 1. Clearly, in the first t time units, i_0 cannot distinguish between the cases of (a) all nodes in the system except i_0 have output bit 1, and (b) only the nodes in N_0 have output bit 1 but all other nodes have output value 0.

By the problem specification, the output value at node i_0 should be different in each of these cases. However, at the first t steps i_0 must deem itself faulty, because of the possibility of case (a) above. Moreover, if the protocol is fault-local, the output stabilization time is $O(f)$ for any f, and i_0 is forced to change its output value to 1 after $O(1)$ steps, and keep it 1 at least through time t, in line with case (a). However, it may later turn out that case (b) is true. The point is that if i_0 removes all traces of its previous output value in the first t steps, i becomes, in effect, an additional faulty node, which might adversely affect other non-faulty nodes later.[2] The intuitive conclusion is that nodes should not purge 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 stabilizing, needs 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 (while keeping old information), and the other for prudent state stabilization (removing obsolete information). Each node has, in addition to the externally visible output bit, another bit we call 'input bit.' Intuitively, the input bit serves as a long term memory (see Figure 1), in the sense that it is used (infrequently) to save the current value of the (frequently changing) output bit. 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, and no node has a wrong estimate of any input value of any other node. The output bit is computed locally by a simple majority rule over these estimates. 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. In the remainder of this section, we describe each part in detail. Both algorithms use variants of the Bellman-Ford Ford spanning tree algorithm [10]; without loss of generality, we shall assume that the spanning tree is unique.

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. These replicas, called *estimates,*

[2] Interestingly, in [20] 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.

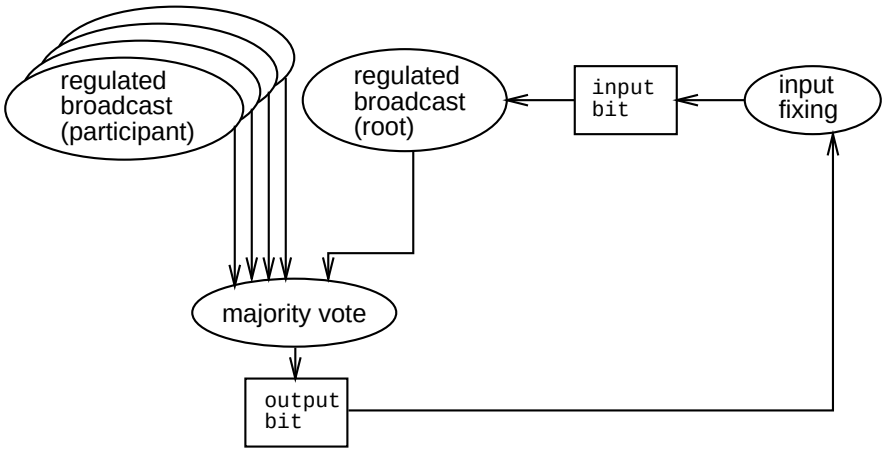

Figure 1: *Schematic structure of the Persistent-Bit protocol at a single node. Each node is the root of one regulated broadcast and a participant of $n-1$ other regulated broadcasts. The output value generated by taking the majority of all values communicated by the regulated broadcast protocol is used by the input fixing protocol, which controls the input bit. Only the output bit is externally observable.*

are used to compute the local output bit by a majority rule. For now, assume that input bits at correct nodes never change (we prove this later). Under this assumption, 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 are estimates of the input values at faulty nodes. We remark 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.

The RB protocol avoids this problem by using the old trick of slowing down to avoid accidents (see e.g. [24, 1, 4]). It particular it is very similar to the *power supply* technique, suggested independently in [2], where it is also shown to be a rather general technique for self stabilization, that can be applied even in unidirectional networks. 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 (this is done by exposing the internal value only after an additional copy step). 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 (but does not harm the other trees). Because of the speed difference, a wrong value cannot reach too far before it is eliminated.

Pseudo-code for output stabilization is presented in Figure 2. The underlying broadcast protocol

is based on the standard Bellman-Ford technique. For all nodes i, j, the $\texttt{value}_i[j]$ entry is the current estimate of node i for the input bit of node j, and $\texttt{value}_i[i]$ is the input bit for node i. The majority function ignores $-$ values and outputs 0 in case of a tie. To slow down the propagation of information, the `value` and `dist` arrays are not read by the neighbors; instead, one step after the computation, each node copies these arrays to other buffers called `t_value` and `t_dist`, which are read by the neighbors. This additional relaying slows down the flow of information to half speed.

We now analyze the RB protocol. Since each RB tree works independently of the others, we may consider a single representative tree rooted at a non-faulty node j. (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.*

- *A node i is* correct with respect to $j \neq i$ *if all the following conditions hold true.*
  - $\texttt{value}_i[j] = \texttt{value}_j[j]$
  - $\texttt{dist}_i[j] = \mathbf{dist}(i,j)$
  - $\texttt{dist}_i[j] = \min\{\texttt{dist}_k[j] + 1 \mid (k,i) \in E\}$
  - $\texttt{dist}_i[j] = \texttt{dist}_{\texttt{parent}_i[j]}[j] + 1$
- *A node i is* correct with respect to itself *if* $\texttt{dist}_i[i] = 0$ *and* $\texttt{parent}_i[i] = \texttt{null}$.
- $\texttt{parent}_i[j]^t$ *is the t-th ancestor of node i in j's tree obtained inductively by following t* $\texttt{parent}[j]$ *pointers starting at i. By convention,* $\texttt{parent}_i[j]^0 \stackrel{\text{def}}{=} i$.

The following lemma is a minor variant of the standard self-stabilization property of the Bellman-Ford algorithm. (The argument can be found in [10]; a similar argument for the self stabilizing case appears in [13, 16, 7].)

Lemma 3.2 *Let $t \geq 0$. If* $\texttt{value}_j[j]$ *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 $inconsistent_k(j)$, given in Figure 2, 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* $\texttt{value}_i[j] \neq \texttt{value}_j[j]$. *Then for some $0 \leq t \leq f$, there exists a node $k = \texttt{parent}_i[j]^t$ such that $inconsistent_k(j)$ holds.*

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

$\texttt{value}_i[V]$	: array of $\{0, 1, \perp\}$	(local estimates)
$\texttt{parent}_i[V]$	: array of $\mathcal{N}(i) \cup \{\texttt{null}\}$	
$\texttt{dist}_i[V]$	: array of $\{1, \ldots, \mathcal{D}\} \cup \{\infty\}$	
$\texttt{t\_value}_i[V]$	: array of $\{0, 1, \perp\}$	(readable by neighbors)
$\texttt{t\_dist}_i[V]$	: array of $\{1, \ldots, \mathcal{D}\} \cup \{\infty\}$	(readable by neighbors)
$\texttt{output}_i$	$\in \{0, 1\}$	

Shorthand

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

Actions for Node i

at every pulse:

$\texttt{parent}_i[i] \leftarrow \texttt{null}$ (write constant values)
$\texttt{t\_dist}_i[i] \leftarrow 0$
$\texttt{t\_value}_i[i] \leftarrow \texttt{value}_i[i]$
for each $j \in V \perp \{i\}$ **do**
  $\texttt{t\_value}_i[j] \leftarrow \texttt{value}_i[j]$ (expose value from previous pulse)
  $\texttt{t\_dist}_i[j] \leftarrow \texttt{dist}_i[j]$
  (compute new values from neighbors)
  $\texttt{parent}_i[j] \leftarrow k \in \mathcal{N}(i)$ such that $\texttt{t\_dist}_k[j] = \min\{\texttt{t\_dist}_l[j] \mid l \in \mathcal{N}(i)\}$
  $\texttt{dist}_i[j] \leftarrow \texttt{t\_dist}_{\texttt{parent}_i[j]}[j] + 1$
  $\texttt{value}_i[j] \leftarrow \texttt{t\_value}_{\texttt{parent}_i[j]}[j]$
  if $\mathit{inconsistent}_i(j)$ **then** (propagate erasure immediately)
    $\texttt{parent}_i[j] \leftarrow \texttt{null}$
    $\texttt{t\_dist}_i[j] \leftarrow \texttt{dist}_i[j] \leftarrow \infty$
    $\texttt{t\_value}_i[j] \leftarrow \texttt{value}_i[j] \leftarrow \perp$
$\texttt{output}_i \leftarrow \text{majority}\{\texttt{value}_i[j] \mid j \in V\}$

Figure 2: *Code for output stabilization at processor i.*

Proof: By contradiction. Define P to be the set of nodes $\{\texttt{parent}_i[j]^t \mid 0 \leq t \leq f\}$. Suppose that for all $k \in P$ we have that $\mathit{inconsistent}_k(j)$ is false. By definition and transitivity, $\texttt{value}_i[j] = \texttt{value}_k[j]$ 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 the correction wave: all wrong estimates disappear quickly.

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

Proof Sketch: For $t > 2 \cdot \mathbf{depth}(i)$, 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. It is easy to see that 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 *Starting from an arbitrary state with fault number $f < n/2$, if the input values at all non-faulty nodes do not change, then after $\min(2 \cdot \mathbf{diam}, 4f + 2)$ steps the output stabilizes, that is, all output values are equal to the input values of non-faulty nodes.*

Proof: If $4f+2 \geq 2 \cdot \mathbf{diam}$ the claim follows from Lemma 3.2. So suppose now that $4f+2 < 2 \cdot \mathbf{diam}$. Fix a node i, and let B denote the set of all nodes at distance at most $2f$ from i. Since $f < n/2$, we have that $|B| \geq 2f + 1$ (including i). Since there are at most f faulty nodes, it follows that the majority of nodes in B is non-faulty. Suppose without loss of generality that $\texttt{value}_j[j] = 0$ for all non-faulty nodes j. Then, by Lemma 3.2, after $t \geq 2 \cdot 2f + 1$ time units, $\texttt{value}_i[j] = 0$ for at least $f + 1$ nodes $j \in V$. 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

$4f + 1$ 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 fault mending algorithm of [19], but the algorithm of [19] 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 $1 + 2 \cdot \min(\mathbf{depth}(i), 2f)$ 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 even 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 bit at node i is correct if it remains unchanged for $T(i, f) \stackrel{\text{def}}{=} 1 + 2 \cdot \min(\mathbf{depth}(i), 2f)$ time units. The problem can therefore be reduced to (1) constructing a 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 compute $T(i, f)$.

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 a faulty node changes its input bit 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 stabilizing timer which does not expire prematurely at non-faulty nodes. Since we assume (to bound the time needed for state stabilization) that faults cease to occur, it is trivial to have a counter which is incremented at each step. Computing the value of $T(i, f)$ in which the timer expires seems harder, since it may depend on faulty nodes.

We solve this difficulty by upper-bounding $T(i, f)$ with $\mathbf{depth}(i)$. This can be done by trivially using the *a priori* upper bound $\mathcal{D}$ on **diam**, but in this case the state stabilization time is propor-

tional to $\mathcal{D}$. We are interested in the scenario where the network topology is unknown ahead of time, and the actual diameter may be significantly smaller than $\mathcal{D}$. In out input-fixing protocol, we include a little stabilizing protocol which preserves (at non-faulty nodes) a certain invariant even before stabilization.

State for Node i

$\texttt{parent}_i[V]$	: array of $\mathcal{N}(i) \cup \{\texttt{null}\}$	
$\texttt{dist}_i[V]$	: array of $\{1, \ldots, \mathcal{D}\} \cup \{\infty\}$	
$\texttt{depth}_i$	$\in \{0, \ldots, \mathcal{D}\} \cup \{\infty\}$	
$\texttt{count}_i$	$\in \{0 \ldots 2 \cdot \mathcal{D} + 1\}$	
$\texttt{output}_i$	$\in \{0, 1\}$	(from Fig. 2)
$\texttt{input}_i$	$\in \{0, 1\}$	(called $\texttt{value}_i[i]$ in Fig. 2)

Actions for Node i

at every pulse:

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

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

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

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

$\texttt{depth}_i \leftarrow \max\{\texttt{dist}_i[j]\}$

if $\texttt{depth}_i$ or $\texttt{output}_i$ are modified **then** $\texttt{count}_i \leftarrow 0$

if $\texttt{count}_i \geq 2 \cdot (\texttt{depth}_i + 1)$ **then**

  $\texttt{input}_i \leftarrow \texttt{output}_i$

  $\texttt{count}_i \leftarrow 2 \cdot \mathcal{D} + 1$

else $\texttt{count}_i \leftarrow (\texttt{count}_i + 1) \bmod (2 \cdot \mathcal{D} + 2)$

Figure 3: *Input fixing protocol: code for processor i.*

Pseudo-code for the input fixing protocol is presented in Figure 3. 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 $\texttt{output}_i$ 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 $\texttt{depth}_i$. As we show, at that point, either $\texttt{depth}_i \geq \mathbf{depth}(i)$ or i is faulty. In any case, i can safely assign $\texttt{output}_i$ to be its input bit $\texttt{value}_i[i]$.

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,* $\texttt{dist}_i[j] \geq \min(t, \textbf{dist}(i,j))$.

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* $\texttt{depth}_i$ *remains unchanged for* $\texttt{depth}_i + 1$ *time units. Then* $\texttt{depth}_i \geq \textbf{depth}(i)$.

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

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

Theorem 3.8 *Starting from an arbitrary state with fault number $f < n/2$:*

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

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

(3) The state of the input fixing protocol stabilizes in $O(\textbf{diam})$ steps; that is, in $O(\textbf{diam})$ steps the network is in a legal state.

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 3), the following must hold true when j changes its input value:

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

Since j is correct, in the start state we have $\texttt{value}_j[j] = \texttt{output}_j$, and hence at some point during the execution j changed its output value. At that step, the code dictates that $\texttt{count}_j$ was reset to 0. Therefore, since $\texttt{count}_j$ is incremented at each step, we must have that $\texttt{depth}_j$ and $\texttt{output}_j$ did not change their values for at least $2 \cdot \texttt{depth}_j + 1$ time units. It follows from Corollary 3.7 that $\textbf{depth}(j) \leq \texttt{depth}_j$, and hence at least $2 \cdot \textbf{depth}(j) + 1$ 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 $\texttt{output}_j$ 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 $2 \cdot \textbf{depth}(j) + 1$ time units, $\texttt{output}_j$ is correct. By Lemma 3.2 and the code, after at most $2 \cdot \textbf{depth}(j)$ additional time units, the input value will be changed to the correct value. The statement follows from the fact that for all $j \in V$, $\textbf{depth}(j) \leq \textbf{diam}$.

(3) By (1) and (2), the input values stabilize after $O(\mathbf{diam})$ time units. The result follows from the standard 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. The scenarios described in the proofs hold even if the specific network topologies used in the proof are known to the protocol.

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, were hit by a batch of faults 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 batch of faults 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 the global state does 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 $0 \leq p \leq n$ 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 4). 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 (see Figure 4).

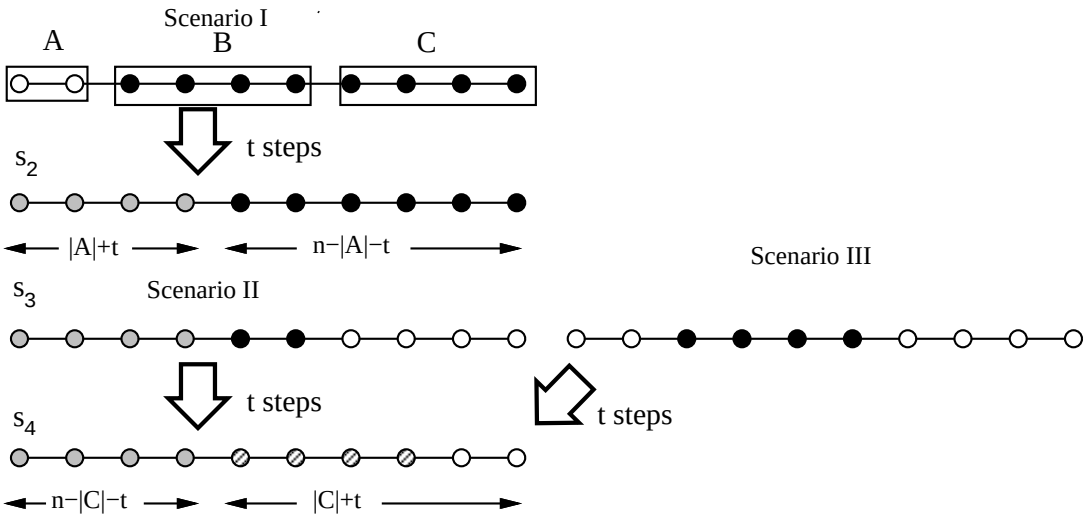

Figure 4: *Example considered in the proof of Lemma 4.2. White nodes are in local state s_0, black nodes are in local state s_1, and gray nodes are in other different states.*

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 batch of faults 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 batch of faults 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 batch of faults which changed their state to be $s_1|_B$. The problem specification requires that the outcome for this scenario is 0.

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 $s_4 = s'_4$. We treat independently the left $|A| + t$ nodes and the right $|C| + t$ nodes: this separartion is justified when these parts do not intersect, which in turn is guaranteed when $t \leq \frac{n-|A|}{4}$ (since $|C| = \frac{n-|A|}{2}$). First, consider the left $|A| + t$ nodes. Note that $s_4|_{L(|A|+t)} = s_2|_{L(|A|+t)}$ by the quiescence of s_2; also, $s_2|_{L(|A|+t)} = s'_4|_{L(|A|+t)}$ because the nodes in $L(|A| + t)$ take the same first t steps in Scenarios III and I. It follows that $s_4|_{L(|A|+t)} = s'_4|_{L(|A|+t)}$. Next, consider the right $|C| + t$ nodes. Note that $s_4|_{R(|C|+t)} = s'_4|_{R(|C|+t)}$ because the nodes in $R(|C| + t)$ take the same first t steps in Scenarios II and III. Finally, since $t \leq \frac{n-|A|}{4}$ the right and left parts are disjoint, and since the nodes in $V \setminus (L(|A| + t) \cup R(|C| + t))$ are quiescent in both Scenarios II and III, we have that $s_4 = s'_4$, 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 $o(\mathbf{diam})$, even if the network topology is known.*

5 The General Transformer

In this section we show how to transform a given algorithm for a general non-reactive problem to a fault-local 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. Moreover, a thorough treatment of reactive problems (rather than non-reactive problems dealt with here) is needed to give a solutions to the initialization problem. We assume that inputs are given by the environment already replicated.

Intuitively, the statement below views the state before the output is computed as faulty, and computing the output is viewed as recovering from faults. The idea is to view the computation as consisting of three stages: first the inputs are replicated at all nodes in the system (as mentioned above, this is outside the scope of this paper) in the second stage, which is done instantly, the outputs are computed; and in the last stage, which is considered to be long-lived, the replicated inputs are guarded against state corruption.

We do not think that this 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 stabilizing protocol $\mathcal{P}'$ which solves Π with output stabilization time $O(f)$ and state stabilization time $O(\mathbf{diam})$, even if f is unknown.*

Proof: We outline the transformed protocol $\mathcal{P}'$. 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 Saving in 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 batch of transient faults. 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 the regulated broadcast algorithm such that each node has a `value` array with only $2F+1$ entries: this is sufficient in order to compute the correct result when the number of faults is at most F. To do this, all we need is that only $2F+1$ nodes will be the sources of regulated broadcast trees. This set can be hardwired in the protocol or somehow dynamically computed. The input fixing protocol becomes much simpler: we needs only to implement a timer which counts up to $4F$, and there is no need to compute the depth of the node.

The change in the code is minimal. In Figure 2, the last **for** statement should be not for every $j \in V$, but rather for every root node j. For completeness, Figure 5 below contains the code for the input fixing part for the case of a known upper bound on F.

State for Node i

$\texttt{count}_i$	$\in \{0 \ldots 4 \cdot F\}$	
$\texttt{output}_i$	$\in \{0,1\}$	(from Fig. 2)
$\texttt{input}_i$	$\in \{0,1\}$	(called $\texttt{value}_i[i]$ in Fig. 2)

Actions for Node i

at every pulse:

if $\texttt{output}_i$ is modified **then** $\texttt{count}_i \leftarrow 0$
if $\texttt{count}_i \geq 4 \cdot F$ **then**
    $\texttt{input}_i \leftarrow \texttt{output}_i$
    $\texttt{count}_i \leftarrow 4 \cdot F$
else $\texttt{count} \leftarrow (\texttt{count} + 1) \bmod (4 \cdot F)$

Figure 5: *Input fixing known F: code for node i.*

It is not difficult to see that with these modifications, the stabilization times are as claimed. (Use the same argument as the one used for Lemma 3.2 for the regulated broadcast; the equivalent of Lemma 3.4 will not be necessary; then use a standard Bellman-Ford argument for the input-

fixing part.) The space required at a node is proportional to F. The saving is significant if F is much smaller than **diam**. It should be clear from Theorem 4.3, however, that such a variant of the protocol cannot withstand $f > F$ faults with the given state stabiliziation time.

Acknowledgment

We thank Oded Goldriech for many very useful comments and the anonymous referees for their help in clarifying many important points.

References

[1] Y. Afek, B. Awerbuch, S. A. Plotkin, and M. Saks. Local management of a global resource in a communication network. In *28th Annual Symposium on Foundations of Computer Science, White Plains, New York*, Oct. 1987.

[2] Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *Proc. of the 8th ann. ACM-SIAM Symposium on Discrete Algorithms*, pages 111–120, 1997.

[3] Y. Afek and S. Dolev. Local stabilizer. In *Proceedings of the 5th Israel Symposium on Theory of Computing and Systems*, June 1997.

[4] Y. Afek and E. Gafni. Distributed algorithms for unidirection networks. *SIAM J. Comput.*, 23(6), Dec. 1994.

[5] 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.*

[6] R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM J. Comput.*, 26(2):539–556, June 1997.

[7] 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).

[8] 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.

[9] 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.

[10] D. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1992.

[11] I. Chlamtac and S. Pinter. Distributed node organization algorithm for channel access in a multihop dynamic radio network. *IEEE Trans. Computers*, C-36(6):728–737, June 1987.

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

[13] S. Dolev. Optimal time self-stabilization in dynamic systems. In *Proc. 7th Workshop on Distributed Algorithms*, pages 160–173, 1993.

[14] 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.

[15] 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.

[16] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, Aug. 1990.

[17] 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.

[18] 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.

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

[20] 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.

[21] A. Mayer, M. Naor, and L. Stockmeyer. Local computation on static and dynamic graphs. In *Proc. of the 3rd Israel Symp. on Theory and Computing Sys.*, 1995.

[22] M. Naor and L. Stockmeyer. What can be computed locally? In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, California*, pages 184–193, 1993.

[23] 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.

[24] G. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4:758–762, October 1982.