Adaptive Stabilization of Reactive Protocols

Shay Kutten[1] and Boaz Patt-Shamir[2]

[1] The Technion, Haifa 32000, Israel. kutten@ie.technion.ac.il
[2] Tel Aviv University, Tel Aviv 69978, Israel. boaz@eng.tau.ac.il

Abstract. A self-stabilizing distributed protocol can recover from any state-corrupting fault. A self-stabilizing protocol is called *adaptive* if its recovery time is proportional to the number of processors hit by the fault. General adaptive protocols are known for the special case of function computations: these are tasks that map static distributed inputs to static distributed outputs. In *reactive* distributed systems, input values at each node change on-line, and dynamic distributed outputs are to be generated in response in an on-line fashion. To date, only some specific reactive tasks have had an adaptive implementation. In this paper we outline the first proof that *all* reactive tasks admit adaptive protocols. The key ingredient of the proof is an algorithm for distributing input values in an adaptive fashion. Our algorithm is optimal, up to a constant factor, in its fault resilience, response time, and recovery time.

1 Introduction

Self-stabilizing distributed systems (sometimes abbreviated stabilizing systems) recover from a particularly devastating type of fault: state-corrupting faults. A state corrupting fault may flip arbitrarily the bits of the volatile memory in the affected nodes; such faults are tricky, since the local state of each processor may seem perfectly legal, and only a global view can indicate that the state is actually corrupted. The model of stabilizing systems, that entails the idea of state-corrupting faults, is an abstraction of *all* transient faults: if a system is self-stabilizing, then it can recover from any transient fault, so long as its code remains intact. Many systems today are implicitly designed to be stabilizing, at least in some sense. For example, one of the popular techniques used by practitioners to achieve partial stabilization is the time-out mechanism: the idea is that each piece of information is stamped with a "time to live" attribute, which says when does this particular piece of information expire. When executed properly, this approach ensures that stale state will eventually be flushed out of the system. But to ensure correctness, the duration of the timeout (and hence the stabilization time) is proportional to the worst-case cross-network latency (see, e.g., the spanning tree algorithm of Perlman [27]).

Another approach to make a system stabilizing, championed mainly in the theoretical community, is the *global reset* method (see, e.g., [11]). In this approach, the idea is that a special stabilizing mechanism monitors the system for illegal states, and whenever an inconsistency is detected, it invokes a special stabilizing reset protocol that imposes some legal global state on the system. The

best reset protocols offer stabilization with relatively low space and communication overhead [10, 9, 11], but at a price of inherently high stabilization time—the cross-network latency time. Still, the time complexity of reset is better than the time-out approach for the following reason. On one hand, time-outs expire after a fixed pre-determined amount of time, which must obviously be the *worst-case* cross-network latency. On the other hand, the stabilization time of a reset protocol is proportional to the *actual* cross network latency when the reset is invoked, and the actual time is typically much better than the worst-case time.

Nevertheless, the global reset approach was not widely adopted in practice. Intuitively, the reason for that is that reset-based stabilization is too "twitchy": the slightest disturbance to the consistency of the system may trigger a system-wide service outage (or "hiccup") for a non-negligible amount of time, which is clearly undesirable. In response to this shortcoming, a new approach, called *adaptive* protocols, has recently taken the focus of attention in this research area [23, 18, 22, 21, 3]. Informally, a stabilizing system is called adaptive if its recovery time depends on the severity of the fault, measured by the number of processors whose state was corrupted. For example, the adaptive system proposed in [21] has the following property: if the state of f nodes is arbitrarily corrupted, then the correct output is recovered everywhere in $O(f)$ time.

However, most previous results for adaptive systems were limited to *distributed function computation*: in this model, it is assumed that each node has a constant input, and the task is to output a fixed function of the inputs. For example, the input at each node may be the (non-changing) weights of its incident edges, and the output is a (non-changing) minimum spanning tree of the network graph. This model, while appropriate for some types of applications, does not capture the full generality of distributed *reactive* systems [24]. In a reactive system, the environment injects new inputs to the system from time to time, and the system is required to produce new output values depending on the given inputs, in an on-line fashion. More precisely, a specification of a reactive task consists of all possible inputs, and for each input, all possible outputs. Unfortunately, the only known results for adaptive reactive protocols either restrict the fault model in a significant way, or they give ad-hoc solutions for specific problems (see below).

In this paper we outline the first proof that general reactive systems can be implemented by an adaptive protocol. More specifically, we consider the following setting. We assume that we are given a synchronous system, where in each round, nodes exchange messages and perform some local computations. In each step, the environment (that models users, or other interacting applications) may input a value to each node, and expects output values to appear at the nodes. The *reactive task* specifies what value each node is required to output at each step. The output values are typically a function of (possibly remote) input values.

If each node had all input values locally available, then it could compute its required output at each step. Thus, the key ingredient in our general implementation is an adaptive algorithm for distributing input values, which we describe in detail in this paper. This primitive task simply requires all nodes to eventually

output a value input at a distinguished source node. Our algorithm is *adaptive* and optimal, up to a constant factor, in the following measures.

- *Response time*: the elapsed time between inputting a value and changing the relevant output values.
- *Recovery time*, which consists of the following two measures [18, 21]. *Output stabilization* is the time it takes until the outputs stabilize to correct values after a fault; and *state stabilization*, which is the time until the system completely recovers internally from a fault (meaning that it is prepared to sustain another fault).
- *Resilience*: the severity of faults from which the algorithm fully recovers.

Resilience is measured in terms of agility [12]. To explain this concept, consider the following situation. Suppose that immediately after a value is input at a node, a fault occurs, and the state of that node is corrupted. Clearly, there is no way for the system to recover the input value in that case, since all its traces may have been completely wiped out. More generally, assume that a message can traverse one link in one time unit. Now, if a value is input at a node v at time t, and at time $t_f \geq t$ all nodes in distance $t_f - t$ from v are hit by a fault, then, by the same reasoning, the input value may be irrecoverable. In this paper, we consider protocols that can recover from such faults—assuming that they occur sufficiently late in the game (so that the protocol gets the chance to replicate the input value elsewhere). The notion of agility makes this intuition concrete. Formal definition is provided in Section 2.

As mentioned above, in this paper we focus on solving the basic building block problem we call *Stabilizing Value Distribution* (abbreviated SBD). We observe that any reactive task can be reduced to SBD, and therefore, we only outline this reduction in this paper. The reduction is straightforward, albeit inefficient in terms of communication and memory overhead. In this paper we concentrate not on the algorithmic ideas of the reduction (the one we propose is rather simple), but rather in the existential proof that any implementable reactive specification has an adaptive solution.

Contributions of this paper. Our main technical contribution is the first adaptive protocol for the basic building block of SBD. Our protocol is also self-stabilizing, and has optimal agility (up to a constant factor). As a corollary, the presented protocol shows that theoretically, any reactive task admits a self-stabilizing adaptive implementation. Our protocol complements results of [22, 21] dealing with non-reactive tasks. There, it is assumed that all inputs and outputs are initially replicated at all nodes. A fault may corrupt some replicas, and the task is to recover the original values. In a way, this task is self-stabilizing, adaptive consensus. The question "how to perform the initial replication?" in a self-stabilizing and adaptive manner is answered in the current paper.

Related work. The study of stabilizing protocols was initiated by Dijkstra [14] for the task of token passing. General algorithms started with *reset-based* approaches

[20, 4, 10, 5]. 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. (The correct state may either be some agreed upon fixed state, or a state that is in some sense "close" to the faulty state [16].) The best reset protocols in terms of time are given in [9], where the stabilization time is proportional to the diameter of the network. Logarithmic space protocols are given in [4, 10], and a randomized constant-space protocol is given in [19]. An extensive survey of self-stabilization is offered in [15].

The idea of adaptive protocols (with variants called fault-local, local stabilizing, or fault containing) is treated in [22, 23, 18, 21, 6, 7], all of them non-reactive.

In [21] it is proven that if a fault hits f nodes, then the output stabilization time is $\Omega(f)$ time units; and that the state stabilization time may be as large as the network diameter even for a small number of faults. This establishes the optimality of our algorithm in the recovery time complexity measures.

In [13], an adaptive protocol for specific task of token passing on a ring is presented. The only general solution for adaptive stabilization of reactive tasks is [3], but it uses a much weaker fault model. Specifically, in [3] it is assumed that the effect of a fault at a node is to change the local state to one chosen *uniformly at random*. This unique assumption allows the protocol to detect faults locally with high probability, by artificially "padding" local state spaces with many identifiable, unreachable local states. The assumption that faults drive the system to a uniformly chosen random state means that with high probability, each affected node is put in one of these bogus states, and hence nodes can locally detect whether their state is legal. We stress that the model of [3] is a fundamental departure from the self-stabilization model (used in the current paper): in self-stabilization, the heart of the difficulty is that faults are *not* locally detectable; in [3], the focus is on local correction.

The technique of *core*, used in this paper, was proposed in [12] for broadcast with error confinement (in error confinement, the goal is to allow only nodes that were directly hit by a fault to err). Roughly speaking, the idea of the technique of core is to perform broadcast in stages such that nodes receiving the broadcast message in a certain stage, consult with a set of the nodes that received it in a previous stage (called the current "core"). In [12], nodes joined the core using a specific rule that ensured error confinement. In this paper we use a different rule, that ensures adaptivity (the rule of [12] is not adaptive).

Another tool we use here is a technique that is becoming rather popular in adaptive protocols, namely that error recovery messages travel faster than other kinds of messages. This technique, called *regulated broadcasts*, appeared first in [21, 2] and was used also, e.g., in [7, 13].

Paper organization. In Section 2 we formalize the model and introduce some notation. In Section 3 we present our algorithm for SBD, and in Section 4 we analyze its properties. Finally, in Section 5, we briefly discuss extensions of the basic result.

2 Preliminaries

2.1 System Model

The system topology is represented by an undirected connected graph $G = (V, E)$, where graph nodes represent processors (also termed network nodes) and edges represent communication links. The number of the nodes is denoted by $n = |V|$. The distance (in the number of edges) between nodes $u, v \in V$ is denoted by $\mathsf{dist}(u, v)$. The diameter of the graph is denoted by diam. We denote

$$\mathsf{ball}_v(d) = \{u \mid \mathsf{dist}(v, u) \leq d\}$$

for $d \geq 0$ (thus $\mathsf{ball}_v(0) = \{v\}$). For $v \in V$, we define $\mathcal{N}(v) = \mathsf{ball}_v(1) - \{v\}$, called the *neighbors* of i. We assume that the network topology is fixed and known to the nodes.

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. Included in each local state are distinguished *input* and *output registers*, visible to the *external environment*. The environment can take two types of actions: input injection, i.e., assign values to input registers, and fault injection, i.e., arbitrarily change the state of an arbitrary set of nodes. The nodes whose states are modified by a fault injection action are said to be *faulty*. By convention, we denote the set of faulty nodes by F, their number by $f = |F|$, and the time of the fault by t_f. To abstract the fact that fault injections are infrequent, we assume without loss of generality that there is just one fault (in fact, another fault may occur after the system has stabilized from the previous one). We say that the faulty nodes were *hit* by a fault at time t_f. If a node v was not hit by a fault in a time interval I, we say that v is *I-intact*.

We assume that the system is synchronous, namely the execution proceeds in rounds, where in each round, each processor sends messages to its neighbors, receives messages, and does some local computation. In this paper we do not restrict message sizes, which allows us to abstract the underlying communication mechanism by assuming that actions may depend also on the state of neighboring nodes (this is justified, e.g., in [4, 17]). Thus, in each step, each node reads its own variables and the variables of its neighbors, and then changes its local state according to the actions specification. As a convention, we denote the location of variables using subscripts, and their time using parentheses; for example, $\mathtt{B}_v(t)$ refers to the value of the variable `B` in node v at time t. Time is measured by the number of synchronous steps.

As is usually assumed, a state corrupting fault may change only volatile state, but not code nor constants such as the node's unique identity `ID`.

2.2 Tasks and Problem Statement

An *input assignment* (respectively, *output assignment*) is a mapping from node names to a given input domain (resp., output range). An *input assignment history* (resp., *output assignment history*) for time t is a set of input assignments

(resp., output assignments), one for each time step $0, 1, \ldots, t$. A reactive *task* (or *problem*) is specified by a function mapping each time step to a binary relation over the input and output histories. This means that a reactive problem says what are the possible inputs, and for each input, what is the required output. A reactive problem is said to be *solved* by a given algorithm if in any execution of the algorithm, at each time step t, the sequence of values taken by the input and output registers satisfy the mapping specified by the problem for time t.

Standard techniques (based on the full-information protocol) show that one can reduce any reactive problem to the following basic building block problem.

> **Stabilizing Value Distribution problem** (SBD).
> Each node v has a single output register denoted by $\texttt{out}_v$. A special node called *source*, denoted by s, has, in addition, an input register denoted by $\texttt{B}_s$. At time 0, the environment writes an *input value* $\texttt{B}_s(0)$. The requirement is that eventually, $\texttt{out}_v$ holds $\texttt{B}_s(0)$ for each node v.

The requirement is to be fulfilled even though at some unknown time $t_f > 0$, some unknown subset F of the nodes is corrupted arbitrarily. We denote $f = |F|$.

2.3 Agility

Consider a fault that occurs at time t_f. We say that a value input to the system at time t_0 is *ρ-recoverable* if only a minority of the nodes in distance $\rho(t_f - t_0)$ from the origin of the value are affected by the fault, for some $0 \leq \rho \leq 1$. An environment is said to be *ρ-constrained* if all inputs are ρ-recoverable. For a given ρ, a system is said to have *agility* ρ if it eventually outputs the correct outputs when run on a ρ-constrained environment. For example, the protocol for SBD in which the source repeatedly broadcasts its input value has 0 agility: once the source is hit by a fault, it may never recover to produce correct output values. On the other hand, a system with agility 1 can recover from any fault so long as the majority of nodes that *potentially* could have heard about the input value remains intact.

3 The Algorithm

In this section we present an algorithm for the SBD problem. We first review the technique of regulated broadcast, introduced in [21].

3.1 Regulated Broadcasts

Regulated Broadcast (abbreviated RB) is an adaptive protocol to distribute and maintain a value under conditions that are more favorable than those studied in the current paper. Specifically, the problem of regulated broadcast is identical to SBD with the crucial difference that *the source is never faulty*. The value at node v of the regulated broadcast rooted at s is called the *vote* of s at v. In our

implementation, the RB protocol will be initiated by many nodes and thus we will have many independent *instances* of the RB protocol running in parallel. We identify instances by their root node.

A vote x of node u at node v is called *authentic* if x was indeed communicated by u. The protocol presented in [21] ensures the following properties.

Lemma 1. *If the fault occurs at some time t_f, and it affects f nodes then:*

- *By time $t_f + 2f$, any vote received by any node is authentic.*
- *Each node u starts receiving, at time no later than $t_f + 2 \cdot \mathsf{dist}(u, v)$, authentic votes at every time step.*

If no fault occurs, each node u starts receiving, by time $2 \cdot \mathsf{dist}(u, v)$, authentic votes at every time step (the RB *protocol is assumed to start at time* 0*).*

Note that in the case the above Lemma, hence no faults occur during time interval $(t_f, t_f + 2 \cdot f]$, by the assumption that no additional batch of faults occur until the system stabilizes from the current batch. (Otherwise the algorithm still stabilizes, but is not required to be time adaptive).

As a consequence, the RB protocol allows every non-faulty node v to verify whether a value communicated to it by an RB protocol is authentic. We formalize this verifiability property in the following lemma for later reference. (It follows directly from the RB properties.)

Lemma 2. *Suppose that node v receives the same vote from a node u during $\mathsf{dist}(u, v)$ consecutive steps. Then this vote is authentic.*

3.2 Algorithm for SBD

Overview. The algorithm presented here expands the ideas of [12], presented there for the more limited task of error confinement. At time 0, the source s gets the input value and starts a broadcast to all other nodes. The idea is to quickly, but carefully, create replicas of the original input value. The algorithm should be quick, in the sense that it should create many replicas, or otherwise it will have low agility. On the other hand, the algorithm should be careful in the sense that it should try to make sure that new replicas have the correct values, or otherwise its action would only be to amplify the effect of the original fault.

Consider a node v that receives a message that is supposed to be sent by a remote node u. There are two difficulties to be answered. First, the message may have never been sent by u: a faulty node between u and v may have altered its contents, or even fabricated it completely from scratch. And second, even if u has indeed sent the message, there is no reason for v to adopt the contents of the message blindly, as u may be faulty, or it have been fooled earlier!

The basic approach we take to overcome these difficulties is to slow the system down, so that the protocol will have enough time to make sure that its actions are sound. Specifically, the first difficulty is circumvented by using regulated broadcast, that allows each node to verify the authenticity of each message it receives (cf. Lemma 2). The second concern is addressed by a special variant of the core technique, whose main property is the following.

Definition 1 (Core Invariant). *At each time step t there exists a set of nodes* $\mathsf{core}(t)$ *such that the majority of the votes of nodes in* $\mathsf{core}(t)$ *is exactly the original input value.*

The central idea of the algorithm is that once a node v has verified the votes of a majority of the current core, then it can (1) set its output correctly, and (2) join the core itself, and start disseminating the correct value. To make the algorithm adaptive, the core-joining rule is based on time as follows.

Definition 2. *A node v joins the core at time t after verifying the authenticity of the votes of a majority of the nodes of some previous core* $\mathsf{core}(t')$ *where* $t' = \mathsf{P}t$ *for some constant* $\mathsf{P} < 1$.

As we shall see, this rule leads to asymptotically optimal agility while maintaining adaptivity. Intuitively, this rule allows core to grow during the time it takes v to consult the nodes of $\mathsf{core}(\mathsf{P}t)$, thus improving the agility.

We remark that in the algorithm of [12], a different rule is used. The Core Invariant is maintained inductively by forcing each node v to verify directly that the values of all the nodes in some core were received in v before letting v join the core. Definition 2 is simpler and not operational.

Algorithm Description. The algorithm works as follows. The source node s sleeps until the environment writes the input value in $\mathtt{B}_s$, and then the source initiates an RB protocol rooted at the source. We say that $t_0^s = 0$. Each other node $v \neq s$ sleeps until v receives an RB message, at a step denoted by $t_0^v > t_0^s$. Non-source nodes start their own instance of the RB protocol when a certain condition is met (see below).

To complete the specification of the algorithm, we need to explain when does a node start its RB, what is the value each node broadcasts, and what is the value it writes in its output register. We describe the algorithm for a generic node v. Lemma 1 motivates the following concepts.

Definition 3. *Node u is said to have a* stable *vote at node v at time t, denoted by* $\mathsf{stable}_v(u,t)$*, if v receives the same value from the* RB *rooted at u for at least* $\mathsf{dist}(u,v)$ *consecutive time units in the time interval* $[t - 4\mathsf{dist}(u,v), t]$*. A node set A is said to be* verified *at node v at time t, denoted by* $\mathsf{verified}_v(A)$*, if there is a majority of nodes of A with identical stable votes at node v at time t.*

At each time step t, v does the following. We use $\alpha = 0.107$ and $\beta = 2\alpha$. We define $\mathsf{core}(t) \equiv \mathsf{ball}_s(\alpha t)$.

Algorithm Disseminate

(1) Participate in the currently active RB instances. Each instance of the RB protocol carries the identity of its root u, the current distance from the root, and the value of $\mathtt{B}_u$.
(2) Set $\mathtt{out}_v$ to be the majority of the current votes of RBs of nodes in $\mathsf{core}(t)$.
(3) If $\mathsf{verified}(\mathsf{core}(\mathsf{P}t))$ is true, then set $\mathtt{B}_v$ to the majority of values in $\mathsf{core}(\mathsf{P}t)$ and start an RB of $\mathtt{B}_v$.

(4) If $\mathtt{B}_v$ is defined, then continue the execution of the RB protocol rooted at v, disseminating the value of $\mathtt{B}_v$ (which may be different than the value used in the previous step).

For Steps 2 and 3, note that $\mathsf{core}(t)$ can be locally computed by v for any t since the topology is known to v and hence the distance to each node is known. For Step 3, note that $\mathsf{verified}$ can be computed by virtue of Lemma 2: this is done by locally counting the number of steps since the last change of the value arriving from the RB rooted at u.

4 Analysis

Intuitively, an execution unfolds as follows. At any given time, the core nodes execute an RB rooted in each of them. Each node that received and verified the votes from a majority of some *previous* core becomes a core member itself. The core expansion rate is α, i.e., at time t all nodes at distance αt from the source are in the core. The interesting point is that the core cannot grow too fast: to make it grow fast, nodes must learn very quickly about the previous core, which they cannot due to physical distance that forces long delays to ensure verifiability.

We start with the following lemma concerning verifiability.

Lemma 3. *Consider an* RB *protocol rooted at a node u, and suppose that $\mathtt{B}_u$ remains fixed during a time interval $[t_1, t_2)$. Then, for every node v with $\mathsf{dist}(u,v) \leq \frac{t_2-t_1}{6}$ such that v is $[t_1,t_2)$-intact, we have that $\mathsf{stable}_v(u,t_2)$.*

Proof. There are two cases to consider. If no fault occurs before time $t_1 + 3 \cdot \mathsf{dist}(u,v)$, then by Lemma 1 v receives $\mathtt{B}_u(t_1)$ at least during the interval $[t_1 + 2 \cdot \mathsf{dist}(u,v), t_1 + 3 \cdot \mathsf{dist}(u,v)]$. Hence, $\mathsf{stable}_v(u,t)$ holds at least in the interval $[t_1 + 3 \cdot \mathsf{dist}(u,v), t_1 + 6 \cdot \mathsf{dist}(u,v)]$, and the lemma is satisfied in this case. If a fault occurs at time $t_f < t_1 + 3 \cdot \mathsf{dist}(u,v)$, then by Lemma 1 we have that by time $t_f + 2 \cdot \mathsf{dist}(u,v)$, v starts receiving $\mathtt{B}_u(t_1)$ uninterrupted, and therefore $\mathsf{stable}_v(u,t)$ starts holding no later than time $t_f + 3 \cdot \mathsf{dist}(u,v) < t_1 + 6 \cdot \mathsf{dist}(u,v)$, and we are done in this case too. □

Intuitively, the algorithm is feasible only if the time interval from $\mathsf{P}\tau$ to τ is large enough for the votes of all the nodes in $\mathsf{core}(\mathsf{P}\tau)$ become stable at nodes in $\mathsf{core}(\tau) - \mathsf{core}(\tau - 1)$. The following technical lemma will be used to show that for a certain choice of P and α, feasibility is achievable.

Lemma 4. *Let $t_0 > 0$, $0 < \alpha < 0.107$, $\mathsf{P} = 2\alpha$, and $\tau \geq t_0/\mathsf{P}$. Then $\mathsf{P}\tau + 6(t_0/2 + \alpha\mathsf{P}\tau) \leq \tau$.*

Proof. By assumptions, we have

$$0 \;\geq\; \mathsf{P}\tau + 6(t_0/2 + \alpha\mathsf{P}\tau) - \tau \;\geq\; 4\mathsf{P}\tau + 6\alpha\mathsf{P}\tau - \tau \;=\; 8\alpha\tau + 12\alpha^2\tau - \tau\ .$$

Dividing by τ and solving the quadratic equation $12\alpha^2 + 8\alpha - 1 \leq 0$ for α, we get that it is satisfied for $(-\sqrt{7/36} - 1/3) \leq \alpha \leq (\sqrt{7/36} - 1/3) \approx 0.107$. □

The following lemma proves the feasibility of the algorithm.

Lemma 5. *Let τ be a time and v be any node with $\mathsf{dist}(s,v) \leq \tau/2$ such that v is $[\mathsf{P}\tau,\tau]$-intact. Assume that for all $u \in \mathsf{core}(\mathsf{P}t) - F$ at all times $\mathsf{P}\tau \leq t < \tau$, the following holds: (1) $\mathtt{B}_u(t) = \mathtt{B}_s(0)$, and (2) node u is the root of an* RB *protocol disseminating $\mathtt{B}_u(t)$. Then $\mathsf{verified}(\mathsf{core}(\mathsf{P}\tau))$ is true at v at time τ.*

Proof. Consider a node v and let $u \in \mathsf{core}(\mathsf{P}\tau)) - F$. By Lemma 1, $\mathsf{dist}(s,v) \leq t_0^v/2$. By definition of core, $\mathsf{dist}(s,u) \leq \alpha\mathsf{P}\tau$, and hence, by the triangle inequality, $\mathsf{dist}(u,v) \leq t_0^v/2 + \alpha\mathsf{P}\tau$. By assumption, u started its RB no later than time $\mathsf{P}\tau$. Therefore, by Lemma 3, we have that $\mathsf{stable}_v(u,t)$ holds for all $t \geq \mathsf{P}\tau + 6(t_0/2 + \alpha\mathsf{P}t)$. In particular, by Lemma 4, it holds for $t = \tau$. Hence, by time τ, $\mathsf{stable}_v(u,\tau)$ holds. Since this is true for all $u \in \mathsf{core}(\mathsf{P}\tau) - F$, and since $|F| < |\mathsf{core}(\mathsf{P}\tau)|$, we may conclude that $\mathsf{verified}(\mathsf{core}(\mathsf{P}\tau))$ is true at v at time τ, as required. □

We now show that if the environment is sufficiently constrained, then the values disseminated by algorithm are correct.

Lemma 6. *If $f < |\mathsf{core}(\mathsf{P}t_f)|/2$ then for all times τ and for all nodes $v \in \mathsf{core}(\tau)$ such that v is $[t_0^v, \infty]$-intact we have that $\mathtt{B}_v(\tau) = \mathtt{B}_s(0)$.*

Proof. (Sketch.) We prove that the following invariant holds for all times τ:

1. If $v \in \mathsf{core}(\tau) - \mathsf{core}(\tau - 1)$ and v is $[t_0^v, \infty]$- intact, then $\mathtt{B}_v(\tau) = \mathtt{B}_s(0)$.
2. If $v \in \mathsf{core}(\tau)$, v is $[t_0^v, \infty]$- intact, and $\mathtt{B}_v(\tau) = \mathtt{B}_s(0)$, then $\mathtt{B}_v(\tau + 1) = \mathtt{B}_s(0)$.

Clearly, the invariant implies the lemma. To prove the invariant, suppose, for contradiction, that it does not hold, and let τ be the first time the invariant is violated. First, note that $\tau > 0$ since the invariant holds trivially for $\tau = 0$. Now, if the invariant is violated at time τ, then there exists a node $v \in \mathsf{core}(\tau)$ such that v is $[t_0^v, \infty]$-intact, and such that at time τ, we have $\mathtt{B}_v(\tau) \neq \mathtt{B}_s(0)$. We first claim that $\mathsf{verified}(\mathsf{core}(\mathsf{P}\tau))$ holds at v. This follows from the fact that by the minimality of τ and the algorithm, all nodes in $\mathsf{core}(\mathsf{P}\tau)$ continuously broadcasts their value, and therefore Lemma 5 guarantees that $\mathsf{verified}(\mathsf{core}(\mathsf{P}\tau))$ holds. Moreover, by the definition of $\mathsf{verified}$, the value of the votes of these nodes at v was stable, and hence it is authentic by Lemma 2.

Finally, we note that since the invariant holds at time $\mathsf{P}\tau$, we have that the votes of each node $u \in \mathsf{core}(\mathsf{P}\tau) - F$ is correct, i.e., $\mathtt{B}_u(\tau') = \mathtt{B}_s(0)$ for all $\tau' < \tau$. It remains to show that these votes are the majority of the votes of $\mathsf{core}(\mathsf{P}\tau)$. To see that, we consider two possible cases. If there are no faulty nodes, or if $t_f > \tau$, then we are done. Otherwise, $t_f \leq \tau$, and hence, by assumption, $|F| < \dfrac{|\mathsf{core}(\mathsf{P}t_f)|}{2} \leq \dfrac{|\mathsf{core}(\mathsf{P}\tau)|}{2}$. □

We can now summarize the properties of our algorithm. For conciseness, we treat the case of no faults as $t_f = \infty$.

Theorem 1. *Let $\rho = 2\alpha^2$. Suppose that a fault hits a set F of f nodes at time $t_f \leq \infty$. If $f < |\mathsf{ball}_s(\rho t_f)|/2$, then*

1. *There exists a time* $T = t_f + O(f)$ *such that* $\mathtt{out}_v(t) = \mathtt{B}_s(0)$ *for all* $t > T$ *and all* $v \in \mathsf{ball}_s(\alpha t)$.
2. *There exists a time* $T = \frac{1}{\mathsf{P}} \min\{t_f, 2\mathsf{diam}\}$ *such that* $\mathtt{B}_v(t) = \mathtt{B}_s(0)$ *for all* $t > T$ *and* $v \in \mathsf{ball}_s(\alpha t)$.
3. *For all* $t < t_f$ *and* $v \in \mathsf{ball}_s(\alpha t)$ *we have that* $\mathtt{out}_v(t) = \mathtt{B}_v(t) = \mathtt{B}_s(0)$.

Proof. Let $v \in \mathsf{ball}_s(\alpha t_f)$. To prove Part 1, first note that, by Lemma 1, there exists a time $T_1 = t_f + O(f)$, such that by time T_1, v receives only authentic votes. We claim that the votes equal to $\mathtt{B}_s(0)$ is a majority among the votes received at v. By Lemma 6 and the assumption on the number of faults, there are at most f incorrect votes from nodes in $\mathsf{core}(t) \supseteq \mathsf{ball}_s(\rho t_f)$ (for any $t \geq t_f$). On the other hand, $2f < |\mathsf{ball}_s(\rho t_f)| \leq |\mathsf{ball}_s(\alpha t)|$, and hence $|\mathsf{ball}_v(2f+1) \cap \mathsf{ball}_s(\alpha t)| \geq 2f+1$. Therefore, by Lemma 1, there exists a time $T_2 = t_f + O(f)$ such that at least $f + 1$ correct votes arrive. The claim follows for $T = \max\{T_1, T_2\}$.

We prove Part 2 of the theorem. Let $T = \frac{1}{\mathsf{P}} \min(t_f, 2 \cdot \mathsf{diam})$, and consider any $t \geq T$. By Lemma 6, this part of the theorem holds for every node $v \in \mathsf{core}(t)$ such that v is $[t_0^v, \infty]$-intact. We prove the claim for $v \in F \cap \mathsf{ball}_s(\alpha t)$ such that $t_f \geq t_0^v$. Namely, $\mathsf{dist}(s, v) \leq \frac{t_f}{2} \leq \frac{t\mathsf{P}}{2}$. By the choice of T, v is $[\mathsf{P}t, t]$-intact. Finally, by Lemma 6, for all $u \in \mathsf{core}(t) - F$ we have that $\mathtt{B}_u(t) = \mathtt{B}_s(0)$. Thus we can apply Lemma 5 and conclude that $\mathsf{verified}(\mathsf{core}(\mathsf{P}t))$ holds at v at time t. This means that v assigns to $\mathtt{B}_v(t)$ the majority vote of all the nodes of $\mathsf{core}(\mathsf{P}t)$. These votes are authentic, by Lemma 2. By the assumption on f, and since $t \geq t_f$, for a majority of these authentic votes the following holds: each comes from some node u that is $[t_0^u, t]$ intact. By Lemma 6 the votes of this majority are correct. The claim follows.

The proof of Part 3 is similar to the proof of Part 2. □

5 Conclusion

In this paper we introduced the first protocol that implements broadcast in an adaptive way. Due to lack of space, we can only sketch here a few applications and extensions of this SBD protocol. First, as already mentioned, the SBD protocol implies the existence of an adaptive solution for any adaptive task. More details are given in [1]. Briefly, in synchronous systems, where the topology is known in a advance, the idea is as follows: in each round, the input at each node is considered to be the root of a new instance of SBD, and the output can be computed locally since all inputs are eventually available locally.

Second, we note that the SBD protocol replicates its root bit *everywhere*. It is possible to trade this maximal fault resiliency for better complexity, by parameterizing the protocol to have some prescribed amount of replications. Third, let us mention that the assumption that the network is synchronous can be lifted using known techniques (e.g., [2] gives an asynchronous RB protocol).

Finally, let us stress once again that while in this paper we demonstrated the existence of an adaptive solution, much work remains to be done in making such a solution practical in terms of computational complexity.

References

1. Longer version of this paper iew3.technion.ac.il//zipped/kp00.ps.
2. Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. of Theoretical Computer Science*, 1998(3), Dec. 1998.
3. Y. Afek and S. Dolev. Local stabilizer. *JPDC*, 62(5):745–765, 2002.
4. Y. Afek, S. Kutten, and M. Yung. The local detection paradigm and its applications to self-stabilization. *Theor. Comput. Sci.*, 186(1-2):199–229, 1997.
5. A. Arora and M. Gouda. Distributed reset. *IEEE T. Comp.*, 43(9):1026–1038, 1994.
6. A. Arora and H. Zhang. GS^3: scalable self-configuration and self-healing in wireless networks. In *Proc. 21st PODC*, pages 58–67, July 2002.
7. A. Arora and H. Zhang. LSRP: Local stabilization in shortest path routing. In *Proc. 2003 Int. Conf. on Dependable Systems and Networks (DSN)*, 2003.
8. B. Awerbuch, I. Cidon, I. Gopal, M. Kaplan, and S. Kutten. Distributed control for PARIS. In *9th PODC*, 1990.
9. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *Proc. 25th STOC*, pages 652–661, 1993.
10. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *32nd FOCS*, pages 268–277, Oct. 1991.
11. B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset. In *Proc. 8th WDAG*, pages 326–339. 1994.
12. Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *22nd PODC*, pages 33–42, June 2003.
13. J. Beauquier, C. Genolini, and S. Kutten. Optimal reactive k-stabilization: the case of mutual exclusion. In *18th PODC*, pages 209–218, May 1999.
14. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17(11):643–644, November 1974.
15. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
16. S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. of Theoretical Computer Science*, 1997(4), Dec. 1997.
17. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *9th PODC*, 1990.
18. S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *15th PODC*, May 1996.
19. G. Itkis and L. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th FOCS*, pages 226–239, Nov. 1994.
20. S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *10th PODC*, Quebec City, Canada, Aug. 1990.
21. S. Kutten and B. Patt-Shamir. Time-adaptive self-stabilization. In *16th PODC*, pages 149–158, 1997.
22. S. Kutten and D. Peleg. Fault-local distributed mending. In *14th PODC*, 1995.
23. S. Kutten and D. Peleg. Tight fault locality (extended abstract). In *36th FOCS*, pages 704–713, 1995.
24. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 3:609–678, 1993.
25. J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the ARPANET. *IEEE Trans. Comm.*, 28(5):711–719, May 1980.
26. J. Moy. OSPF version 2, Apr. 1998. Internet RFC 2328.
27. R. Perlman. *Interconnections.* Addison-Wesley Publishing Co., 2nd edition, 2000.