

# Distributed Error Confinement

## Extended Abstract

Yossi Azar  
Dept. of Computer Science  
Tel Aviv University  
Tel Aviv 69978  
Israel  
azar@cs.tau.ac.il

Shay Kutten  
Dept. of Industrial Engineering  
Technion  
Haifa 32000  
Israel  
kutten@ie.technion.ac.il

Boaz Patt-Shamir<sup>\*</sup>  
HP Cambridge Research Lab  
One Cambridge Center,  
Cambridge MA 02142  
USA  
Boaz.PattShamir@HP.com

### ABSTRACT

We initiate the study of error confinement in distributed applications, where the goal is that only nodes that were directly hit by a fault may deviate from their correct external behavior, and only temporarily. The external behavior of all other nodes must remain impeccable, even though their internal state may be affected. Error confinement is impossible if an adversary is allowed to inflict arbitrary transient faults on the system, since the faults might completely wipe out input values. We introduce a new fault tolerance measure we call *agility*, which quantifies the strength of an algorithm that disseminate information, against state corrupting faults.

We study the basic problem of broadcast, and propose algorithms that guarantee error confinement with optimal agility to within a constant factor, even in asynchronous networks when the topology is unknown. These algorithms can serve as building blocks in more general reactive systems. Previous results in exploring locality in reactive systems were not error confined, and relied on the assumption (not used in current paper) that the errors hitting each node are probabilistic, such that a faulty node itself, or its neighbor, can detect the node faulty.

The main algorithm uses the novel *core bootstrapping* technique, that seems inherent for voting in reactive networks; its analysis leads to an interesting combinatorial problem. The technique and the analysis may be of independent interest

### 1. INTRODUCTION

One key difference between centralized and distributed systems is that in distributed systems, faults may hit only a part of the system. To achieve error confinement we to benefit from the fact that many nodes may have not been hit. This intuition was explored before for the case of non-reactive systems (but error confinement, defined below, was not achieved). This becomes harder in distributed

<sup>\*</sup>On leave from Department of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel.

systems that are required to propagate information, e.g. communicating new input of one node to remote nodes. The difficulty stems from the fact that the propagation may amplify the effect of a fault by spreading wrong information across the system. This, in effect, causes the receiving nodes to become faulty too. That is, their output and messages differ from the case that no faults hit the system. This phenomenon acts against the attempt to benefit from the fact that many of the nodes may not have been hit by the original faults. The main technical contribution of this paper, in Subsections 3.3, 3.4, is an algorithm that prevents this amplification effect.

Previous papers avoided this difficulty, and addressed issues that are still to be faced if this difficulty does not exist. For example, in [2] it was assumed that every faulty node can detect itself faulty since it was also assumed that the nature of the faults was probabilistic. Thus, a faulty node knows not to spread the faults. The task of that paper was then to recover from the faults fast, if the number of faults was small. (Error confinement, defined below, was not addressed, but is immediate in that model.) A similar principle is used in [26] (there the fault can be detected by a neighbor). Similarly, [16] avoided the spreading issue by addressing only the state that the propagation is not needed; it handled problems remaining after the information was already somehow spread correctly, and only then faults occurred. Thus, again, the nodes corrupted by faults were not required to spread their values (and the faults). The task remaining was just to hold some “special” kind of a consensus in which each non-faulty node voted the already spread value and the faulty nodes may have voted another. That consensus was “special” in the sense that it was in a self stabilizing model, and in the sense that it was required to be fast when the number of faults was small.

Some fault-resilient protocols deal with faults by allowing arbitrary behavior until recovery is complete (intuitively, declaring a temporary “state of emergency”). In papers such as [16, 2, 17, 13] the question is how to shorten this period (if the number of faults is small). The problem addressed in this paper is how to devise systems that keep the faulty information masked from the external user as much as possible, even during the recovery period.

Let us be more specific (see Section 2 for formal definitions). We consider the model of a distributed system that executes some reactive task, i.e., the environment inputs values at nodes and reads outputs from nodes. The requirement is specified as a predicate over the sequences of input and output values. We consider transient faults, i.e., faults that eventually leave the system. This is modeled by assuming that a fault may hit a set of nodes by arbitrarily modi-

ifying their state, but this may happen at most once in the execution. (If additional faults occur “enough time” after the first fault, the error confinement property is preserved in the algorithms presented below; on the other hand, if additional faults occur “soon after” the first, the algorithms still self stabilize, but may not be error confined). Intuitively, a system is said to have the error confinement property if in any execution, the observable input and output values meet the specification, where the only possible exceptions are nodes directly hit by a fault. Note that it is not possible to guarantee a better behavior, since a local fault can trivially violate the specification by changing the value of a local output variable. On the other hand, it is good enough for some purposes. For example, low-level error-confined system can be used to build higher level applications which are error-confined, as we show in this paper.

The specific problem we study is *broadcast*, where a value is input at one of the nodes and the task is that eventually, all other nodes will output that value exactly. The problem of broadcast is interesting in our context for two reasons. First, the essence of broadcast is dissemination of information, in apparent contrast to the idea of error confinement, since the operation of the protocol might contaminate non-faulty nodes with bad information. And secondly, broadcast can be viewed as a complete problem for reactive problem: intuitively, if all nodes know all inputs, then each node can compute its output locally.

Previous papers that handled self stabilizing broadcast did it in a context where the value of the broadcast source node cannot be corrupted. For example, in self stabilizing topology update (e.g. [11]) a broadcast source broadcasts the current status of its link as it knows it at that point in time; its “knowledge” is, by definition, correct. Another example- in [16] every node broadcasts its “vote”, which may be a result of a corruption, but needs to be broadcast nevertheless (if the corrupted were the minority of the votes then the consensus yielded the right result). In Subsections 3.1 we construct an error confined broadcast algorithm for the case that the source is non-faulty. This is used as a building block in our main algorithm that handles the reactive case in Subsections 3.3, 3.4.

Error confinement is not always possible, because if a value is input at a node, and an error hits that very node before it sent out any message, then the input value may have no trace whatsoever in the system, making recovery impossible. On the other hand, it is known how to recover from faults that hit any minority of the nodes, if they occur when the input value is already safely mirrored in all nodes [16]. This motivates us to introduce the measure of *agility*, which quantifies the resilience of reactive algorithms as a function of time. Intuitively, we consider faults that may hit only a minority of the nodes in a ball around the origin of the input value. An algorithm is more agile if that ball grows more quickly, i.e., the agility of the algorithm measures *how quickly do we lift the restriction on the faults*. For example, an algorithm that cannot recover from a corrupted source has agility 0, and an algorithm that can recover at any time  $t$  from a fault that corrupts only a minority of the nodes in the ball of radius  $\frac{t}{5}$  around the source has agility  $\frac{1}{5}$ . The main technical question addressed in this paper is what is the maximal agility that allows the existence of error-confined systems.

**Our contribution.** In this paper we formalize the concepts of error confinement and agility. Note that agility makes sense also for non-confined algorithms. As a concrete example, we consider the basic primitive of broadcast. We prove that error confinement necessar-

ily entails a slowdown of factor 2 for broadcast. We then present an error-confined algorithm whose agility and speed are within a constant factor even from algorithms that are not error confined.

The main novel technique introduced here is that of *core bootstrapping*. It is useful for preventing the amplification of errors by message propagation. Thus it seems to be useful for other tasks in reactive systems. Our main algorithm uses a specific variant of that method, called *ball core*, that is very natural in the context of broadcast in reactive systems. For algorithms using ball cores, we present matching lower and upper bounds on the agility of any broadcast algorithm. We remark that the latter analysis is related to the cow path problem [22], and may be of independent interest.

Most of the presentation is for the synchronous network model and assuming that the network topology is known, but the final algorithm works in the asynchronous network model with unknown topology.

**Related work.** Error confinement is an important aspect in fault-tolerant software systems. See, e.g., [19, 24] and references therein for the software engineering perspective on fault confinement. Malkhi *et al.* [21] study error confinement in the presence of Byzantine faults. Our approach in modeling faults is similar to the one in [20, 7]. The model of state-corrupting faults is implicit in the seminal work of Dijkstra about *self stabilization* [10]: put in our terminology, a system is called self-stabilizing if even after an arbitrary state-corrupting fault occurs (possibly hitting all nodes), eventually the system starts behaving correctly. Some general algorithmic solutions for making a system self-stabilizing appear in [15, 3, 5], based on the paradigm of *reset* [6]: when an error is detected, a global reset action is invoked, imposing a correct state on the system. These papers do not distinguish between a fault that flipped, say, a single bit in the system, and a massive fault that touches many nodes, and therefore a significant disturbance in service is allowed in both cases. A reduction in the *number* of nodes corrected is achieved in many cases by the reset protocol of [11]. This protocol brings the system to a correct global state that is closest to the faulty state, rather than backing into the same correct global state in every situation. This saves in the number of nodes to correct. Still, this algorithm is not error confined. For example, in the case of a broadcast, a fault in a single node may cause every node to output an incorrect value temporarily, and to refuse to output at some other times (during its participation in the reset). Another direction in limiting the damage caused by faults is correlating the recovery time with the number of faulty nodes. In some problems that are inherently local, a single fault can be corrected immediately, as was pointed out in [8, 11]. More general approaches were proposed in [17, 2, 13, 16]. In [13, 23, 16], a distinction is made between the time it takes the observable output to stabilize, and the time it takes for the internal state to stabilize. We stress that all these works allow for correct nodes to exhibit faulty behavior before stabilization.

Another concept worth mentioning is *snap stabilization* [9]: A system is called snap-stabilizing if its behavior stabilizes to its specification in 0 time. Clearly, snap stabilization is possible only for a certain class of task specifications, that allow every faulty node to be considered externally correct even at the time of the fault. Broadcast does not satisfy this requirement. For those problems that do satisfy this requirement it seems that turning snap stabilizing protocols into error confined ones should be possible.

- An *action* is associated with (a.k.a. occurs in) a *node*. An action may be either *external* or *internal*. An external action is either an *input* or an *output* action. Nodes may also be called “processors,” or “sites.”
- A *behavior* is a sequence of external actions. Given a node  $v$  and a behavior  $\beta$ ,  $\beta_v$  is the *local behavior* of  $v$ , i.e., the subsequence of  $\beta$  that consists only of actions that occur in  $v$ .
- A *task* is a set of behaviors, called *legal behaviors* for the task.
- A *protocol* is a specification of *states* and *transitions*. A state is a vector of *local states*, one local state for each node. Transitions are triples denoted  $s \xrightarrow{a} s'$ , where  $s$  and  $s'$  are states, and  $a$  is an action. An action  $a$  is said to be *enabled* in state  $s$  if  $s \xrightarrow{a} s'$  is a transition for some state  $s'$ . Input actions are enabled in all states. A subset of the states is designated as *initial states*.
- An *execution* of protocol  $P$  is an infinite sequence  $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots$ , where  $s_0$  is an initial state of  $P$ , and  $s_{i-1} \xrightarrow{a_i} s_i$  is a transition of  $P$  for all  $i$ . In a *timed* execution, each action is annotated with its time of occurrence.
- Given an execution  $e$ , its corresponding behavior, denoted  $\beta(e)$ , is the sequence of all external actions in  $e$ .
- A protocol  $P$  implements task  $\Pi$  if its set of behaviors is a subset of the legal behaviors of  $\Pi$ .

**Figure 1:** Actions, behaviors, states, executions etc.: IO Automata formalism.

**Organization of this paper.** In Section 2 we formally define the model and the concepts of error confinement and algorithm agility. In Section 3 we develop an error-confined algorithm for the broadcast problem in the synchronous model and analyze its agility. Subsections 3.3, 3.4 present our main algorithmic contribution. The algorithm in Section 3.3 is presented assuming an novel construct we term *core*, while Subsection 3.4 solves an optimization problem in order to select the best core. (This analysis is related to the cow path problem). In Section 4 we present an extension of the basic algorithm to the asynchronous model. We conclude with some discussion in Section 5. Some additional proofs are provided in the appendix.

## 2. BASIC CONCEPTS

In this section we define the model of computation, the broadcast task, and the key concept of algorithm agility.

### 2.1 A Model for Error Confinement

**General parameters.** The system is modeled as a fixed undirected connected graph  $G = (V, E)$ , where nodes represent processors and edges represent bi-directional communication links. We denote  $|V| = n$ , and the diameter of the graph is denoted  $\text{diam}$ . The distance between two nodes  $u, v \in V$ , denoted  $\text{dist}(u, v)$ , is the minimal number of edges in a path connecting them. Given a node  $v \in V$ , we denote  $\text{ball}_v(r) = \{u \in V \mid \text{dist}(v, u) \leq r\}$ , and call it the *ball* of radius  $r$  around  $v$ .

**Error confinement.** To define error confinement formally, we use IO Automata as our underlying formalism [20]. The standard definitions are summarized in Figure 1. In this paper we consider a single type of faults, called *state corrupting*, that abstracts all transient faults. Such faults are formally defined as follows.

**DEFINITION 2.1.** A state corrupting fault is an action that alters the state arbitrarily in some subset of nodes. The nodes whose

local state was altered are called *faulty*.

In our model, there is at most one fault in an execution, which can span a set of nodes.

The new concept we propose is the following.

**DEFINITION 2.2.** A protocol  $P$  is said to be an error-confined protocol for task  $\Pi$  if for any execution with behavior  $\beta$  (possibly containing a fault) there exists a legal behavior  $\beta'$  of  $\Pi$  such that

- (1) For each non-faulty node  $v$ ,  $\beta_v = \beta'_v$ .
- (2) For each faulty node  $v$ , there exists a suffix  $\overline{\beta}_v$  of  $\beta_v$  and a suffix  $\overline{\beta}'_v$  of  $\beta'_v$  such that  $\overline{\beta}_v = \overline{\beta}'_v$ .

The output stabilization time of a faulty node  $v$  is the time duration of the prefix of  $\beta_v$  that is not included in  $\overline{\beta}_v$ .

The main point in the definition above is that the behavior of non-faulty nodes must be exactly as in the specification: only faulty nodes may have some period (immediately following the fault) in which their behavior does not agree with the specification.

Formally, the broadcast task is defined as follows. (A general task, of course, will include many instances of broadcast.)

#### Broadcast (BCAST)

*Input actions:*  $\text{inp}_s(b)$ , done at node  $s \in V$ , and  $b$  in some set  $D$ . The node  $s$  is called *source*.

*Output actions:*  $\text{outp}(b)$ , required at all nodes  $v \in V$ , where  $b \in D \cup \{\perp\}$ .

*Legal behaviors:* There is at most one  $\text{inp}_s$  action. Each node  $v$  outputs  $\text{outp}(\perp)$  in each step up to some point, and then it outputs  $\text{outp}(b)$  in each step, where  $b$  is the value input by the  $\text{inp}$  action.

For ease of exposition, we abuse notation slightly and use special input and output registers called *mirror*: an input action is equivalent to assigning a value to the  $\text{mirror}_s$  register at the source  $s$ , and similarly the output action at node  $v$  just reads the value of the local  $\text{mirror}_v$  value. (We use the convention that variables are subscripted by their node name.)

Error confined broadcast means that if any non-faulty node outputs a value  $a \neq \perp$ , then all non-faulty nodes may output only  $a$  (or  $\perp$ ), and all nodes must output  $a$  eventually.

**Agility.** As mentioned above, there is no way to maintain error confinement in the face of an arbitrary state-corrupting fault: the fault may hit the source immediately after the input action, leaving no trace of the original input value. However, faults can be overcome if they arrive later, since the source could have communicated the input to some other nodes in the meantime. It is impossible to design an algorithm that will ensure replication to more nodes than those in a certain distance (a *ball* around the source) that depends on the time. If a fault hits the majority of nodes in this ball, ensuring recovery is impossible. The notion of  $\alpha$ -constrained environment formalizes this idea.

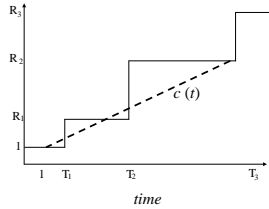
**DEFINITION 2.3.** An environment is called  $\alpha(t)$ -constrained for some function  $\alpha(t)$  and a given system topology if the following

condition holds. Suppose that input is made at node  $s$  at time  $t_0$ , and that a fault occurs at time  $t_f$ . Then the number of nodes hit by the fault is less than  $\frac{1}{2}|\text{ball}_s(\alpha(t_f - t_0))|$ .

The propagation of information in the system is physically limited to within a dynamically growing ball centered at the source, and hence no algorithm can recover inputs if that ball is corrupted. Our definition restricts the faults inside a ball whose radius growth is bounded by the function  $\alpha(t)$ .

**DEFINITION 2.4.** An algorithm for the broadcast problem has agility  $\alpha(t)$  if it has the error-confinement property for all  $\alpha(t)$ -constrained environments. An algorithm is said to have agility  $c$  for a constant  $c$  if it has  $\alpha(t)$  agility for  $\alpha(t) = c \cdot t$  for all  $t \geq 0$ .

Note that while our definition is for error-confinement, it generalizes for any type of fault resilience. The agility of an algorithm, intuitively, defines the maximum rate in which  $\alpha(t)$  grows that still allows the algorithm to be correct.



**Figure 2:** The agility is the rate the constraint on the fault is relaxed to allow more faults

*Example.* This example demonstrate an interesting finding given in Subsection 3.4. If  $\alpha(T_i) = R_i$ , then, at time  $t = T_i - \epsilon$  for some  $\epsilon > 0$ , the constraint on the faults is a ball of radius  $R_{i-1}$  (see example in Figure 2). Thus, if the agility rate is some  $c$ , then  $c \leq \frac{R_i - 1}{T_i}$ . One may expect  $\alpha(t)$  to grow smoothly, rather than maintaining the same value for long periods. That is, one may expect the algorithm to maximize  $\alpha$  in each step greedily. Interestingly, it turns out that the infrequent changes in the value of  $\alpha$  are an inherent property of the optimal algorithm (see Section 3.4). ■

**Synchronous and asynchronous computations.** In most of this paper, we use a simplified model of computation called the *synchronous network model*. In this model, time proceeds in steps, where in each step (called *round* of computation), all nodes first read the state of their neighbors, and then set their own state. This model abstracts the underlying mechanism whose job is to make the state available at neighbors, as well as synchronize their progress. Note that in the synchronous model, states change at discrete steps. When we say “at time  $t$ ,” the interpretation is “in the state between the end of step  $t$  and the start of step  $t + 1$ .” The asynchronous network model is treated in Section 4.

### 3. BROADCAST WITH ERROR CONFINEMENT

The main novel technique in this paper appears in Subsections 3.3, 3.4 where we develop an algorithm for BCAST with error confinement in the synchronous model that works even if the source may

suffer faults. Subsection 3.3 presents the framework of the algorithm, while Subsection 3.4 solves an optimization problem in order to decide the specific parameter. The behavior of the optimum seems interesting in itself.

Before that we construct a building block- an algorithm for BCAST assuming that the source is never faulty, see Subsection 3.1. Recall (from the introduction) that this more limited case (that the source is never faulty) has some similarities to the cases dealt with in some previous papers (though solutions in previous papers were not error confined). This makes its treatment easier, and we use it as a module for our core bootstrapping protocol of Subsections 3.3, 3.4.

We also prove, in Subsection 3.2, a lower bound that says that any algorithm for broadcast must slow down the output by at least a factor of 2, even if the execution is fault-free.

#### 3.1 A building block: Broadcast with a correct source

We first solve BCAST under the assumption that the source is correct. This part is less difficult than the general case of Subsections 3.3, 3.4. Still, this primitive is useful since it has the following partial error confinement property (whether the source was hit by a fault or not): if the algorithm outputs a value at a non-faulty node  $v$ , then the output value is *authentic*, in the sense that it was indeed communicated by the source. (We say “partial” error confinement since, if the source is faulty, the value communicated by the source may be faulty.)

To solve BCAST with a correct source, let us start with the problem of distance computation.

##### Single source distance computation (SSD)

*Input actions:*  $\text{start}_s$ , made at the source node  $s$ .

*Output actions:*  $d$ , where  $d \in \{0, 1, \dots, N\} \cup \{\perp\}$  for some large integer  $N$ .

*Legal behaviors:* Each node  $v$  outputs  $\perp$  in each step up to some point, and then it outputs  $\text{dist}(s, v)$  in each subsequent step.

Without the requirement for error confinement, the Bellman-Ford algorithm solves SSD even in the face of state-corrupting faults (see, e.g., [4]). Informally, the algorithm works as follows. In non-source nodes, the initial value of the output variable  $\text{dist}_v$  is  $\perp$ , and in each step, the node sets its value to be one plus the minimum of the distance variables of its neighbors (where  $\perp$  is treated as infinity). The source node sets its output variable to 0 in each step following the  $\text{start}_s$  action.

Note that the Bellman-Ford algorithm is not error-confined: A non-faulty node sets its distance variable to one plus the minimum of the values of its neighbors, and that minimum may be erroneous. But a simple extension makes Bellman-Ford error-confined. The key observation is that if the distance variable value is  $d$ , and it has not changed for at least  $d$  time units, then the distance of the node from the source is indeed  $d$ . This gives rise to Algorithm A, presented formally in Figure 3. The effect of the input action at the source  $s$  is to set  $\text{dist}_s \leftarrow 0$  in that step and every subsequent step.

We now prove the error confinement property of Algorithm A. Let us assume without loss of generality that the fault occurs at time 0, and that at that point, the state variables have arbitrary values.

<p>State at node <math>v \neq s</math>:</p> <p><math>\text{dist}_v</math>: output distance variable, initially <math>\perp</math></p> <p><math>\text{cand\_dist}_v</math>: internal distance variable, initially <math>\perp</math></p> <p><math>\text{count}_v</math>: counter, initially 0</p> <p>Code at node <math>v \neq s</math>:</p> <pre> if <math>\text{cand\_dist}_v \neq 1 + \min \{ \text{cand\_dist}_u \mid u \text{ is a neighbor of } v \}</math> then   <math>\text{cand\_dist}_v \leftarrow 1 + \min \{ \text{cand\_dist}_u \mid u \text{ is a neighbor of } v \}</math>   <math>\text{count}_v \leftarrow 0</math> else <math>\text{count}_v \leftarrow \min(\text{count}_v + 1, \text{cand\_dist}_v)</math>  if <math>\text{count}_v \geq \text{cand\_dist}_v</math> then <math>\text{dist}_v \leftarrow \text{cand\_dist}_v</math> </pre>
---

**Figure 3:** Algorithm A: Single source distance computation with confined errors.

LEMMA 3.1. At any time  $t \geq 0$ , for any node  $v$ , it holds that  $\text{cand\_dist}_v \geq \min(\text{dist}(s, v), t)$ .

**Proof Sketch:** First, note that the lemma holds trivially for the source node. We proceed by induction on time. For  $t = 0$  the lemma holds since  $\text{cand\_dist}_v \geq 0$  always. For the inductive step, let  $v \neq s$  be any node, and consider time  $t + 1$ . By the induction hypothesis, we have that for every neighbor  $u$  of  $v$ ,  $\text{cand\_dist}_u \geq \min(t, \text{dist}(s, u))$ . If  $t < \text{dist}(s, v)$  for all neighbors  $u$ , we are done, since  $\text{dist}(s, v) = 1 + \min_u \{ \text{dist}(s, u) \}$ , and  $v$  assigns a value which is at least  $t + 1 \geq \min(t + 1, \text{dist}(s, v))$ . If  $\text{dist}(s, u) \leq t$  for some neighbors of  $v$ , let  $u_0$  be the neighbor closest to  $s$ . Note that  $\text{dist}(s, v) = \text{dist}(s, u_0) + 1$ . By induction hypothesis, we have that  $\text{cand\_dist}_{u_0} \geq \min(t, \text{dist}(s, u_0))$ , and since  $\text{cand\_dist}_v \geq \text{cand\_dist}_{u_0} + 1$ , we get that  $\text{cand\_dist}_v \geq \text{cand\_dist}_{u_0} + 1 \geq \min(t, \text{dist}(s, u_0)) + 1 = \min(t + 1, \text{dist}(s, v))$ . ■

LEMMA 3.2. If  $\text{dist}(s, v) \leq d$  for some node  $v$ , then at any time  $t \geq d$ , we have that  $\text{cand\_dist}_v \leq d$ .

**Proof:** By induction on the distance  $\text{dist}(s, v)$ . If  $\text{dist}(s, v) = 0$ , then  $v = s$  the claim follows directly from the code. Assume that the claim is true for all nodes at distance  $\delta$  at all time steps  $t \geq \delta$ , and consider a node  $v$  with  $\text{dist}(s, v) = \delta + 1$ . Let  $u_0$  be a neighbor of  $v$  with  $\text{dist}(s, u_0) = \delta$ . By the induction hypothesis, at time  $\delta$  and onward, we have the  $\text{cand\_dist}_{u_0} \leq \delta$ . It follows from the code that at time  $\delta + 1$  and onward,  $\text{cand\_dist}_v \leq \delta + 1$ , as required. ■

Using the lemmas above, we now analyze the broadcast time. Let the start time of the broadcast be  $t_0$ . If faults occurred before the broadcast delivers the message everywhere then let  $t_f$  be the time of the faults. Otherwise, let  $t_f = t_0$ . We measure the time of the broadcast from  $\max\{t_0, t_f\}$ :

THEOREM 3.3. Algorithm A solves SSD with confined errors and output stabilization time  $2 \cdot \text{diam}$ .

**Proof:** We first prove stabilization. By Lemmas 3.1 and 3.2, we have  $\text{cand\_dist}_v = \text{dist}(s, v)$  for any time  $t \geq \text{diam}$ , at any node

<p>State at node <math>v \neq s</math>:</p> <p><math>\text{mirror}_v</math>: the broadcast value to be output, initially <math>\perp</math></p> <p><math>\text{cand\_mirror}_v</math>: an internal estimate of the output value, initially <math>\perp</math></p> <p><math>\text{cand\_dist}_v</math>: an internal estimate of the distance, initially <math>\perp</math></p> <p><math>\text{count}_v</math>: counter, initially 0</p> <p>Code at node <math>v \neq s</math>:</p> <pre> Let <math>u_0</math> be the neighbor of <math>v</math> such that   <math>\text{cand\_dist}_{u_0} = \min \{ \text{cand\_dist}_u \mid u \text{ is a neighbor of } v \}</math> if <math>(\text{cand\_dist}_v \neq 1 + \text{cand\_dist}_{u_0})</math> or   <math>(\text{cand\_mirror}_v \neq \text{cand\_mirror}_{u_0})</math> then   <math>\text{cand\_dist}_v \leftarrow 1 + \text{cand\_dist}_{u_0}</math>   <math>\text{cand\_mirror}_v \leftarrow \text{cand\_mirror}_{u_0}</math>   <math>\text{count}_v \leftarrow 0</math> else <math>\text{count}_v \leftarrow \min(\text{count}_v + 1, \text{cand\_dist}_v)</math> if <math>\text{count}_v \geq \text{cand\_dist}_v</math> then <math>\text{mirror}_v \leftarrow \text{cand\_mirror}_v</math> </pre>
---

**Figure 4:** Building block: Algorithm B. Broadcast with confined errors assuming the source is non-faulty.

$v$ . Hence,  $\text{count}$  is never reset after time  $\text{diam}$ , and hence, by time  $2 \cdot \text{diam}$  we have  $\text{count}_v \geq \text{dist}(s, v) \geq \text{cand\_dist}_v$ . It follows that by time  $2 \cdot \text{diam}$ , all nodes set  $\text{dist}_v \leftarrow \text{cand\_dist}_v = \text{dist}(s, v)$ , and the system stabilizes as required. Next, we show error confinement. Consider any node  $v$ , and suppose that  $v$  changes the value of its  $\text{dist}_v$  variable at time  $t$ . There are two cases to consider. If the value of  $\text{dist}_v$  is changed less than  $\text{cand\_dist}_v$  time units since the last time the  $\text{cand\_dist}_v$  variable was changed, then clearly the  $\text{count}_v$  variable does not have its intended semantics, and hence node  $v$  is faulty and we are done. So suppose that the  $\text{dist}_v$  variable is set to  $d$  after at least  $d$  time units in which  $\text{cand\_dist} = d$ . By Lemma 3.1 we have that  $\text{dist}(v, s) \geq d$ , and by Lemma 3.2  $\text{dist}(v, s) \leq d$ . The result follows. ■

We remark that Theorem 3.3 holds for a fault that hits any number of nodes at any time, simply because there is no input value.

We now extend Algorithm A to solve the BCAST task. This is done by “piggy-backing” the broadcast value on the distance value, once it is input. The broadcast value becomes externally visible only when the  $\text{dist}$  variable would have become visible in Algorithm A. The algorithm for broadcast with error confinement is formally presented in Figure 4 for non-source nodes. For the source node  $s$ , we have that the  $\text{inp}_s(b)$  action results in assigning  $\text{mirror}_s \leftarrow b$ , and also the source keeps setting  $\text{cand\_dist}_s \leftarrow 0$  and  $\text{dist}_s \leftarrow 0$  in each subsequent step.

THEOREM 3.4. If the source node is non-faulty, then Algorithm B solves BCAST with confined errors, and output stabilization time  $2 \cdot \text{diam}$ .

**Proof Sketch:** Consider a non-faulty node  $v$ , and suppose that it assigns a value to  $\text{mirror}_v$  at time  $t$ . We show that this value is correct. Suppose that at time  $t$ ,  $\text{cand\_dist}_v = d$ . Since  $v$  is non-faulty, we have by the code that at time  $t$ ,  $\text{count}_v = d$ , and that there were at least  $d$  time units during which  $\text{cand\_dist}_v$  did not change.

We claim that  $d = \text{dist}(s, v)$ . First note that if  $d > \text{dist}(s, v)$ , then by Lemma 3.2, by time  $t$  we have that  $\text{cand\_dist}_v = \text{dist}(s, v) <$

$d$ , contradicting the fact that  $v$  did not change its  $\text{cand\_dist}_v$  value. So it must be the case that  $d \leq \text{dist}(s, v)$ . Suppose for contradiction that  $d < \text{dist}(s, v)$ . We show that  $\text{count}_v$  must have been reset by time  $t$  in this case. Let us say that a node  $v$  *consistently depends* on node  $u$  in a given state if  $\text{cand\_dist}_v = \text{cand\_dist}_u + 1$  and  $\text{cand\_mirror}_v = \text{cand\_mirror}_u$ . Nodes  $v_0, v_1, \dots, v_k$  are called a *consistent dependency chain* of  $v_0$  in a given state if  $v_i$  consistently depends on  $v_{i+1}$  for all  $0 \leq i < k$  in that state. Note that if the maximal consistent dependency chain of a node at some state is of length 0, then by code, that node will set  $\text{count}_v \leftarrow 0$  in the next step. Now, consider the  $d$  maximal consistent dependency chains of  $v$ , one chain for each time step  $t-1, t-2, \dots, t-d$ . We claim that at least one of these chains is of length 0, which contradicts the assumption that  $\text{count}_v$  was not reset during this time interval. To see that, first note that the length of the chain at time  $\tau$  is exactly  $d - \text{cand\_dist}_{u(\tau)}$ , where  $u(\tau)$  is the last node in the chain of  $v$  at time  $\tau$ . Since  $d < \text{dist}(s, v)$  by assumption, it follows from the triangle inequality that  $\text{cand\_dist}_{u(\tau)} < \text{dist}(s, u(\tau))$ , and hence, by Lemma 3.1 we have that at time  $t-d+i$  the length of any consistent dependency chain of  $v$  is at most  $d-i$ , and hence there will exist a zero-length chain by time  $t-1$ , as required. The output stabilization time follows directly from the fact that after  $\text{diam}$  time, all  $\text{cand\_dist}$  and  $\text{cand\_mirror}$  variables have the correct values, which in turn follows from Lemmas 3.1 and 3.2. ■

### 3.2 Error confinement implies slowdown

Clearly, under Algorithm B, a node  $v$  outputs a value after  $2 \cdot \text{dist}(s, v)$  time units, even if there are no faults: twice the necessary minimum. The following theorem shows that this slowdown is inherent to error confinement, even if there are no faults (Note that if there are faults, the stabilization time of algorithm B is even somewhat higher.)

**THEOREM 3.5.** *Let  $X$  be an algorithm solving BCAST with error-confinement if the source  $s$  is correct. Then for any non-faulty node  $v$ , the time in which  $v$  outputs a value is at least  $2 \cdot \text{dist}(s, v)$  steps after the input at  $s$ , even if there are no faults.*

**Proof:** Consider a line graph, where nodes are numbered  $0, 1, 2, \dots$ , and let the source be node 0. Consider any node  $i$ . We compare two executions of  $X$ : in execution  $e_0$ , no input is ever made at the source, and in execution  $e_1$ , a value 1 is input at  $s$  at some time  $t_0$ . Let  $t_1$  be the first time in which the execution of node  $v$  differs between  $e_0$  and  $e_1$ . Obviously,  $t_1 \geq t_0 + i$ , since the first difference between  $e_0$  and  $e_1$  occurs at time  $t_0$  at distance  $i$  from node  $i$ . To prove the theorem we claim that algorithm  $X$  cannot output a value at  $i$  before time  $t_1 + i$  in  $e_1$ . This is shown by contradiction: Suppose that  $i$  outputs a value at time  $t_2 < t_1 + i$ . We define another execution  $e'$  as follows. Up to and excluding time  $t_1$ ,  $e'$  is identical to  $e_0$ . At time  $t_1$ , two events occur at  $e'$ : First, an input of value 0 is made at  $s$ ; and second, a fault changes the states of the nodes number  $i-1, \dots, i-(t_2-t_1)$  to be the same as in  $e_1$ . Note that the source is non-faulty because  $i > t_2 - t_1$ . Also note that node  $i$  is non-faulty. It is immediate to verify by induction that the execution of nodes  $i, i-1, \dots, i-(t_2-t_1)+j$  is identical in  $e_1$  and  $e'$  in steps  $t_1, \dots, t_1+j$  since each of these nodes cannot distinguish  $e'$  from  $e_1$  at these times. It therefore follows that node  $i$  will output value 1 in  $e'$ , a contradiction to the error-confinement property that requires all outputs at non-faulty nodes to be the same as the input value. ■

### 3.3 General Error-Confined Broadcast

We now present the main algorithmic contribution of the paper. This algorithm allows for a faulty source, under the assumption that faults may not corrupt the state of a majority of the nodes in  $\text{ball}_s(\alpha(t))$  at time  $t$ . The idea is to apply a bootstrapping technique. While algorithm B used a single source (and had agility zero), the Core-bootstrapping algorithm maintains a dynamically growing set of nodes called  $\text{core}(t)$ , for each time step  $t$  (where  $\text{core}(0) = \{s\}$ ). Each node in the core set broadcasts (using Algorithm B as a primitive) what it believes to be the true value input at  $s$  at time 0. Assuming that no fault ever directly corrupts the majority of the current core, the algorithm ensures that *always*, the majority of values in the core set is correct.

The core grows inductively: A node may join the core if it has “sufficient evidence” to determine that the value it is about to start broadcasting is correct. “Sufficient evidence” here means the set of values broadcast by a complete core set. This is “sufficient” since faults may corrupt only a minority of the core nodes by assumption. Thus, the main task is to select the next core in a way that will lift these constraints on the adversary as fast as possible. This main task is, actually, deferred to the next subsection

For the subtask of correctly collecting the above “sufficient evidence” values, the algorithm uses Algorithm B as a building block, this is why we needed it to be error confined too. This leaves us with the task to design the algorithm in such a way that the assumption on the constraints of the faults is minimal. Specifically, consider the algorithm (with a parametric  $\text{core}(t)$  function) that behaves at node  $v$  in each time step  $t$  as follows:

#### Algorithm Boot:

- (1) Receive broadcasts from all nodes using Algorithm B.
- (2) If Algorithm B locally outputs values from all nodes in  $\text{core}(t-1)$ , set  $\text{mirror}_v$  value to their majority value (otherwise,  $\text{mirror}_v$  retains its previous value).
- (3) If  $v \in \text{core}(t)$ , then broadcast  $\text{mirror}_v$  using Algorithm B.

The algorithm works for any  $\text{core}(t)$  specification, so long as the following condition is satisfied for all nodes  $v$  and time steps  $t$ :

**Feasibility Condition:** If  $v \in \text{core}(t) - \text{core}(t-1)$ , then by time  $t$  node  $v$  received broadcast values  $\text{mirror}_u$  from all nodes  $u \in \text{core}(t-1)$ .

**THEOREM 3.6.** *Let  $\text{core}(t)$  be a function satisfying the feasibility condition, and assume  $|\text{core}(t)| \geq |\text{ball}_s(\alpha(t))|$  for some function  $\alpha(t)$ . Then Algorithm Boot is an error-confined algorithm for broadcast with agility at least  $\min \left\{ \frac{\alpha(t)}{t} \mid t \geq 1 \right\}$  and output stabilization time of  $2 \cdot \text{diam} + \min \{t \mid \text{core}(t) = V\}$ .*

**Proof:** We first prove, by induction on time, that if node  $v$  is non-faulty, then it will never set its  $\text{mirror}_v$  variable to a wrong value. The base case is  $t \leq t_f$ , where  $t_f$  is the time of the fault (possibly,  $t_f = \infty$ ). In this case, the claim follows from the trivial fact that the system is correct before the fault. For the inductive step, suppose that a non-faulty node  $v$  sets its  $\text{mirror}_v$  value at time  $t > t_f$ . By Theorem 3.4, Algorithm B guarantees that the local outputs at  $v$  of the broadcasted values from nodes in  $\text{core}(t-1)$  are authentic. Since less than  $\frac{1}{2}|\text{core}(t_f)|$  nodes are faulty by assumption that  $|\text{core}(t)| \geq |\text{ball}_s(\alpha(t))|$  and since all non-faulty nodes have correct  $\text{mirror}$  values by the induction hypothesis, and since all these

values will arrive at  $v$  by the second part of the feasibility condition,  $v$  will set its `mirrorv` variable to the correct value. This proves the error confinement property.

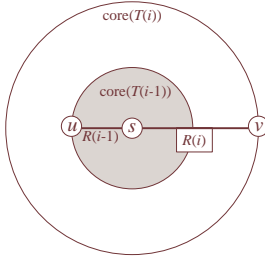
The output stabilization time bound follows by the fact that non-faulty nodes broadcast the correct value and from Theorem 3.4. ■

### 3.4 Ball Core Functions

Algorithm Boot does not specify how the core function is defined, except for the restriction that it is feasible. In this section we focus on ball cores, which match our definition for constrained environments. Ball core functions are defined as follows. In any given time  $t$  we have  $\text{core}(t) = \text{ball}_s(R(t))$  for some  $R(t)$  called the *radius* of the core at time  $t$ . We seek a ball core function that admits the best possible agility.

So fix a ball core function  $\text{core}$ . Let  $\{R_i\}$  denote the sequence of all *distinct* radii of core in increasing order, i.e.,  $R_i < R_{i+1}$  for all  $i$ . Let  $T_i$  denote the first time that  $\text{core}(t) = \text{ball}_s(R_i)$ . See Figure 2. To simplify exposition, and motivated by Theorem 3.5, from now on we normalize the time scale by a factor of 2, which means that the time between the start of Algorithm B at any node  $v$  and the time another node  $u$  has an authenticated value of `mirrorv` is  $\text{dist}(v, u)$  if no faults occur.

The following lemma establishes a strong connection between  $R_i$  and  $T_i$  (see Figure 5).



**Figure 5:** The feasibility condition implies that node  $v$  cannot join the core before  $R_i + R_{i+1}$  time units have passed since node  $u$  joined the core.

LEMMA 3.7. Let  $\text{core}$  be a feasible ball core function. Then for all  $i > 0$ , we have

$$T_i \geq R_i + 2 \sum_{j=1}^{i-1} R_j.$$

Moreover, the best agility for the given radii sequence is attained when equality holds.

**Proof Sketch:** It suffices to prove that  $T_i - T_{i-1} \geq R_i + R_{i-1}$ : The inequality of the lemma then follows by unfolding. To prove the latter inequality, consider any graph with nodes  $s, u, v$  such that  $\text{dist}(s, u) = R_{i-1}$ ,  $\text{dist}(s, v) = R_i$ , and  $\text{dist}(u, v) = R_{i-1} + R_i$ . (A line graph of length  $R_i + R_{i-1}$  does the job.) Now, by construction,  $u \in \text{core}(T_{i-1}) - \text{core}(T_{i-1} - 1)$ , and  $v \in \text{core}(T_i) - \text{core}(T_i - 1)$ . Hence the feasibility condition, and the fact that  $\text{dist}(u, v) = R_{i-1} + R_i$  imply the result. To prove the positive part of the claim, we set  $T_i = T_{i-1} + R_i + R_{i-1}$  inductively. To

see feasibility, note that the distance between the furthest node in the  $i$ th core and the  $(i+1)$ st core is at most  $R_i + R_{i+1}$ . Optimality in case of equality is proven by induction. ■

Hence we have the problem of choosing a sequence of radii  $\{R_i\}$  that maximizes the agility subject to the inequality of Lemma 3.7. Note that the agility of the algorithm is  $\min_i \left\{ \frac{R_{i-1}}{T_{i-1}} \right\}$ : at time  $T_i - 1$ , the core radius is still  $R_{i-1}$ . Interestingly, this problem is closely related to the classical *cow path problem* (see, e.g., [22, 14]), where a cow wanders along a line until it finds food, and the goal of an algorithm is to minimize the ratio between total distance traversed by the cow and the path length between its origin and the food.

To gain some intuition into the problem of choosing the optimal radii sequence, consider the “greedy” rule, where a node  $v$  enters the core at time  $t$  if this is the first time in which  $v$  receives the broadcasts of all nodes in  $\text{core}(t - 1)$ . This means that the radii sequence is  $R_i = i$ . It follows from Lemma 3.7 that the greedy rule leads to the core function  $\text{core}(t) = \text{ball}_s(\sqrt{t})$ , and hence the agility is  $\frac{1}{\sqrt{\text{diam}}}$ . An interesting observation that follows from the next two lemmas is that the number of times the optimal ball core grows (the number of steps in Figure 2) is logarithmic in  $n$ .

The following two lemmas help us choose an optimal sequence of radii. The proofs of these lemmas, given in the appendix, can also serve as new proofs for the cow path problem. The first lemma asserts a lower bound on the agility of any ball core function.

LEMMA 3.8. Let  $R_1, R_2, \dots$  be any positive non-decreasing sequence such that  $R_1 \geq 1$ . Let  $T_1, T_2, \dots$  be a sequence such that  $T_i \geq R_i + 2 \sum_{j=1}^{i-1} R_j$ . Then  $\liminf \frac{R_i}{T_{i+1}-1} \leq \frac{1}{3+2\sqrt{2}}$ .

Lemma 3.8 is stronger than what we need, since it proves the bound for an infinite number of times. The next lemma shows how to pick a sequence of radii that attains the agility lower bound of Lemma 3.8.

LEMMA 3.9. Let  $R_i = \lfloor (1 + \sqrt{2})^{i+1} \rfloor$  for  $i \geq 1$ , and let  $T_i = R_i + 2 \sum_{j=1}^{i-1} R_j$ . Then  $\inf \frac{R_i}{T_{i+1}-1} \geq \frac{1}{3+2\sqrt{2}}$ .

Setting  $\text{core}(t) = \text{ball}_s(\max \{R_i \mid t \leq T_i\})$  in Algorithm bootstrap, where the  $R_i$  and the  $T_i$  values are as given by Lemma 3.9, yields an error-confined algorithm for BCAST with agility  $\frac{1}{3+2\sqrt{2}} \approx 0.172$ , which is optimal for ball cores by Lemma 3.8.

## 4. ASYNCHRONOUS ERROR CONFINED BROADCAST

The presentation of the asynchronous algorithm is deferred to the full paper. Let us here just sketch the extension for Algorithm Boot to the asynchronous model. Using ball cores, we also replace the assumption that the topology is known to the nodes ahead of time by the assumption that only the identity of the source node is known. The extension adds another factor of 2 slowdown to the algorithm performance.

First, let us describe the model. We assume that the basic operations are message send and receive, and that messages can be delayed for an arbitrary (but finite) time in the communication links.

Links can also be garbled by a state corrupting fault, but we assume that in any given time, there is at most one outstanding message that is in transit in every channel. We remark that this model can be generalized (without changing the asymptotic complexity) to allow any number of messages to be stored in a link, so long as this number is bounded by a constant. See [25] for further discussion.

Now, consider Algorithm Boot. The correctness of Algorithm Boot relies on synchrony in two ways: first, Algorithm Boot uses Algorithm B as a subroutine, and the correctness of Algorithm B relies on the assumption that the network is synchronous. And secondly, each node needs to know  $\text{core}(t)$  for any step  $t$ , but there is no well-defined notion of global step number in the asynchronous model.

Let us first describe an asynchronous version of Algorithm B called Algorithm B'. (Similar ideas were used in different contexts in [3, 1].) We start by observing that the idea behind Algorithm B is that before outputting a value, each node ensures that there exists a *causal chain* [18] carrying this value from the source to the destination. This is done by counting communication rounds: sufficiently many rounds without the value changed provide evidence for the existence of a such causal chain. The idea in Algorithm B' is to have *explicit* causal chains, made of messages, that start at the source. The chains form along paths induced by the asynchronous version of the Bellman-Ford algorithm. Counting the number of messages received on a path, each node can obtain a lower bound on the number of messages that must have been sent by each of its ancestors on the tree. Similarly to Algorithm B, Algorithm B' forces each node to reset its counter when changes occur, now including also parent change, or parent counter change. This way, a counter reset propagates down the tree. It turns out that it suffices to count up to  $\text{cand\_dist} + 1$ .

We now address the problem of computing  $\text{core}(t)$  locally. For ball cores, the solution is rather simple: each node knows its distance from the source  $s$  due to Algorithm B'. Consider a node  $v$  such that  $R_{i-1} < \text{dist}(s, v) \leq R_i$ , where  $\{R_i\}$  is the sequence of the core radii. Node  $v$  enters the core (i.e., starts broadcasting) once it receives the mirror values from all nodes in the previous core, namely from all nodes whose distance from  $s$  is at most  $R_{i-1}$ . This can be done in the asynchronous model by requiring each node to broadcast its distance from the source along with its mirror value. To maintain error confinement, we need to address the possible scenario where the distances are not true: All that Algorithm B' ensures is that the value received was indeed sent by the source, but it does not ensure that the *distance* communicated by a source  $u$  of Algorithm B' is indeed  $\text{dist}(s, u)$ . This difficulty is easily solved by applying the slowdown mechanism once again in the following way. If a node  $v$  receives a message from node  $u$  claiming that  $\text{dist}(u, s) = d$ , then using arguments similar to those used in the proof of Theorem 3.4, we can show that delaying the use of  $\text{mirror}_u$  for  $d$  additional messages in which  $u$  continuously claims this distance is sufficient to ensure that this value is correct. This means that  $\text{mirror}_u$  can be used at node  $v$  after  $2 \cdot \text{dist}(u, v) + \text{dist}(u, s)$  messages are received by node  $v$ . By the triangle inequality,  $2 \cdot \text{dist}(u, v) + \text{dist}(u, s) \leq \max(3 \cdot \text{dist}(u, v), 2 \cdot \text{dist}(s, v))$ . The net effect is that when a non-faulty node enters the core, its mirror value is correct, at the price of additional slowdown of most 2. Let B'' denote Algorithm B' with the above extensions.

Finally, we need to address the problem of how to avoid the corruption of correct nodes *after* they enter the core. This is done by

simply not allowing a node to change its mirror value until it hears from all nodes in the system. Specifically, the asynchronous version of Algorithm Boot at a node  $v$  is as follows. Once a message is received from the source  $s$ ,  $v$  starts computing its distance from  $s$  using Algorithm B''. Based on that distance, it computes  $R_{i-1}$  as the maximum core radius smaller than  $\text{dist}(s, v)$ . When Algorithm B'' outputs all mirror values from nodes  $u$  with  $\text{dist}(s, u) \leq R_{i-1}$ ,  $v$  sets  $\text{mirror}_v$  to be the majority of these values, and starts broadcasting that value using Algorithm B''. Node  $v$  may change its mirror value only after it receives (by means of Algorithm B'') the mirror values of *all* nodes in the system.

**THEOREM 4.1.** *If the source node is non-faulty, then Algorithm Async-Boot solves Broadcast with confined stabilization.*

## 5. DISCUSSION

Error confinement, often required in centralized systems, seems to be even more useful in distributed settings. This paper represents only a first step in defining and exploring this notion in distributed systems. It leaves many problems open. Can the agility of our algorithm be improved? Our algorithm uses ball cores. Is this choice optimal? Clearly, there exist topologies and source nodes locations for which there exist better core functions. Another natural choice is that of monotonically growing cores. In some settings, certain nodes are better protected than others, and where this property of nodes may change over time. Our algorithms start with a core that consists of the source only. It may be the case (as in [21]) that the initial core contains multiple nodes.

Another direction is the relaxation of the model assumptions. For example, it is interesting to investigate the case where faults hit in multiple batches. Other questions lie in correlating error confinement to other notions. For example, the notion of time adaptivity [17, 13, 16] restricts the allowed *time* span of the faulty behavior, as a function of the number of faulty nodes. Is there a trade off between such a requirement and that of error confinement, that restricts the *spatial* span of faulty behavior as a function of the number of faults?

We have shown in Theorem 3.2 that error confinement implies a (constant) slow down in the broadcast. This leaves open other questions of overhead implied by confinement. In particular, our algorithms use much more communication resources and memory than non-error confinement algorithms. Can this overhead be reduced? The notion of error confinement presented here applies only to the external behavior. In message passing models it seems that there is no way to prevent the contamination of parts of the state that are not exposed externally. Is this possible in other models?

Finally, there is the question regarding the applicability of the results to general reactive systems. There is a trivial reduction from a general reactive task to broadcast. In this reduction, every node broadcasts its values, and every node can compute the output based on the inputs of all nodes. This, of course, is not a very efficient reduction. It would be interesting to investigate the fault confinement of other problems, as well as the agility that can be achieved for other problems. We believe that our broadcast algorithm will prove a useful primitive for solving such problems.

## 6. REFERENCES

- [1] 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.
- [2] Y. Afek and S. Dolev. Local stabilizer. In *Proceedings of the 5th Israel Symposium on Theory of Computing and Systems*, June 1997.
- [3] 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.*
- [4] 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).
- [5] 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.
- [6] 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.
- [7] M. Breitling. Modeling faults of distributed, reactive systems. In *6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, (FTRTFT 2000)*, pages 58–69. Springer (LNCS 1926), 2000.
- [8] Imrich Chlamtac, Shlomit S. Pinter Distributed Nodes Organization Algorithm for Channel Access in a Multihop Dynamic Radio Network. *IEEE Transactions on Computers* 36(6): 728-737 (1987)
- [9] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (WSS 3)*, pages 78–85. IEEE Computer Society, 1999.
- [10] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17(11):643–644, November 1974.
- [11] 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.
- [12] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. 9th Ann. ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, Aug. 1990.
- [13] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proc. 15th Ann. ACM Symp. on Principles of Distributed Computing*, May 1996.
- [14] M.-Y. Kao, J. Reif, and S. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *Proc. of the 4th ann. ACM-SIAM Symposium on Discrete Algorithms*, 1993.
- [15] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.
- [16] S. Kutten and B. Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220(3):93–111, 1999.
- [17] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proc. 14th Ann. ACM Symp. on Principles of Distributed Computing*, Aug. 1995.
- [18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.
- [19] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, Wien, second revised edition, 1990.
- [20] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1995.
- [21] D. Malkhi, Y. Mansour, and M. Reiter. Diffusing without false rumors: On propagating updates in a byzantine environment. *Theoretical Comput. Sci.*, 2002.
- [22] C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, 1991.
- [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] D. J. Taylor. Practical techniques for damage confinement in software. In *Proceedings of the 1998 Computer Security Dependability and Assurance (CSDA '98)*. IEEE, 1998.
- [25] G. Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.
- [26] I-Ling Yen: A Highly Safe Self-Stabilizing Mutual Exclusion Algorithm. *Information Processing Letters* 57(6): 301-305 (1996).

## APPENDIX

**Proof of Lemma 3.8:** Fix a sequence  $\{R_i\}$ . Let  $S_i = \sum_{j=1}^i R_j$ . Clearly  $T_{i+1} \geq R_{i+1} + 2S_i \geq (1+2i)R_1 > i+1$ . Thus,

$$\begin{aligned} \frac{R_i}{T_{i+1} - 1} &= \frac{R_i}{T_{i+1}} + \frac{R_i}{T_{i+1}(T_{i+1} - 1)} \\ &\leq \frac{R_i}{T_{i+1}} + \frac{1}{T_{i+1} - 1} < \frac{R_i}{T_{i+1}} + \frac{1}{i}, \end{aligned}$$

where the first inequality follows from the fact that  $R_i \leq T_{i+1}$ . We use the fact that for any two sequences  $\{X_i\}_{i>0}$  and  $\{Y_i\}_{i>0}$  we have

$$\liminf (X_i + Y_i) \leq \liminf X_i + \limsup Y_i.$$

Hence

$$\liminf \frac{R_i}{T_{i+1} - 1} \leq \liminf \frac{R_i}{T_{i+1}} + \limsup \frac{1}{i} = \liminf \frac{R_i}{T_{i+1}}.$$

Thus, it is enough to show that  $\liminf \frac{R_i}{T_{i+1}} \leq \alpha$ , where  $\alpha = \frac{1}{3+2\sqrt{2}}$ .

Assume by contradiction that  $\liminf \frac{R_i}{T_{i+1}} > \alpha$ . That is there is some positive  $N$  such that for all  $i > N$  we have

$$\frac{R_i}{T_{i+1}} \geq \beta$$

for some fixed  $\beta > \alpha$ . Using the above with the definitions of  $T_i$  and  $S_i$  we conclude that  $\beta$  is at most

$$\frac{R_i}{2S_i + R_{i+1}} = \frac{S_i - S_{i-1}}{2S_i + (S_{i+1} - S_i)} = \frac{S_i - S_{i-1}}{S_i + S_{i+1}} = \frac{1 - S_{i-1}/S_i}{1 + S_{i+1}/S_i}.$$

Let  $y_i = S_i/S_{i-1}$ . Hence we get that

$$\beta \leq \frac{1 - 1/y_i}{1 + y_{i+1}}.$$

Now define  $z_i = 1 + y_i$  (clearly  $z_i > 1$ ) to get that

$$\beta \leq \frac{1 - 1/(z_i - 1)}{z_{i+1}}.$$

Now, we use the following inequality which holds for any  $t > 1$

$$1 - 1/(t - 1) \leq \alpha t.$$

To prove this inequality we note that it is equivalent (multiplying by  $t - 1$ ) to  $t - 2 \leq \alpha t^2 - \alpha t$  or to  $\alpha t^2 - (\alpha + 1)t + 2 \geq 0$ . The last inequality holds since the determinant  $(\alpha + 1)^2 - 8\alpha = 0$  by our choice of  $\alpha$ .

Using the above inequality we get

$$\beta \leq \frac{1 - 1/(z_i - 1)}{z_{i+1}} \leq \frac{\alpha z_i}{z_{i+1}}$$

Hence

$$z_{i+1}/z_i \leq \alpha/\beta < 1 - \epsilon$$

for some fixed  $\epsilon > 0$ . Hence for large enough  $i$ ,  $z_i$  becomes smaller than 1 which is a contradiction. This completes the proof of the lemma. ■

**Proof of Lemma 3.9:** Let  $\alpha = \frac{1}{3+2\sqrt{2}}$ . Clearly  $\frac{R_i}{T_{i+1}-1} \geq \frac{R_i}{T_{i+1}}$  for all  $i$ . We show that  $\frac{R_i}{T_{i+1}} \geq \alpha$  for all  $i$ .

We first define a sequence  $R'_i$  of real numbers, and then analyze the sequence  $R_i$ . Let  $q = 1 + \sqrt{2}$ . Define  $R'_i = q^{i-1}$  for all  $i > 0$  and  $T'_i = R'_i + 2S'_{i-1} = q^{i-1} + 2(q^{i-1} - 1)/(q - 1)$ . Let  $S'_i = \sum_{j=1}^i R'_j$  as before. With these definitions we have

$$\frac{R'_i}{T'_{i+1}} = \frac{q^{i-1}}{q^i + 2\frac{q^i-1}{q-1}} \geq \frac{q^{i-1}}{q^i + \frac{2q^i}{q-1}} = \frac{1}{q + \frac{2q}{q-1}} = \alpha$$

by our choice of  $q$  (this is, of course, the optimal choice for  $q$  to maximize the ratio). This completes the proof for the sequence of reals  $\{R'_i\}_{i>0}$ .

We now consider the integer sequence  $\{R_i\}_{i>0}$  defined by  $R_i = \lfloor q^{i+1} \rfloor$ . By definition,  $R_i = \lfloor R'_{i+2} \rfloor \geq R'_{i+2} - 1$  for all  $i > 0$ . Let  $S_i = \sum_{j=1}^i R_j$ . Clearly  $S_i \leq S'_{i+2} - (1+q)$ , since we omitted the

first two elements of the sequence, and rounded the other elements down. Hence

$$T_{i+1} = R_{i+1} + 2S_i \leq R'_{i+3} + 2S'_{i+2} - 2(1+q) = T'_{i+3} - 2(1+q).$$

Now, since  $2(1+q)\alpha > 1$  we conclude that

$$R_i \geq R'_{i+2} - 1 \geq \alpha T'_{i+3} - 1 \geq \alpha(T'_{i+3} - 2(1+q)) \geq \alpha T_{i+1}$$

where the second inequality follows from the fact that we proved the lemma for the sequence  $R'_i$  with real numbers. This completes the proof. ■