Title	On Time Adaptivity and Stabilization (Algorithm Engineering as a New Paradigm)
Author(s)	Kutten, Shay
Citation	数理解析研究所講究録 (2001), 1185: 120-129
Issue Date	2001-01
URL	http://hdl.handle.net/2433/64633
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録
1185巻 2001年 120-129

On Time Adaptivity and Stabilization

Shay Kutten

Technion, Haifa 32000, Israel, kutten@ie.technion.ac.il

Abstract:

We study the scenario where a transient fault hit a minority of the nodes in a distributed system by corrupting their state. The notion of *time adaptive*, or of *fault locality* were introduced to take advantage of the fact that the number of faults in such realistic scenarios may be small. A stabilizing distributed protocol is called *adaptive* if its recovery time from a state-corrupting fault is proportional to the number of processors hit by the fault, rather than to the total number of nodes.

We describe time adaptive protocols, and upper and lower bound for their stabilization time, and their fault resilience. We describe basic problems, both in the static case, and in the interactive case, such that more general problems can be reduced to them. We then solve these problems optimally. We describe the main techniques, as well as the new definitions, such as that of the amount of fault resilience that can be obtained, the different kind of stabilization (state stabilization versus output stabilization), and the additional desirable properties for such protocols.

This area is rather new, and a lot of addition research is still needed.

Keywords: Time Adaptive, mending, Distributed Algorithms, Stabilization.

1 Introduction

Stabilizing distributed systems recover from a particularly devastating type of faults: state-corrupting faults. During a state corrupting fault, the bits of the volatile memory in an affected processor may be arbitrarily flipped. Their resilience against such faults make stabilizing systems highly desirable, as it generalizes resilience against any kind of transient fault. Naturally, stabilization is quite expensive in terms of computational resources. For example, one of the common methods to make a system stabilizing is the *reset* paradigm, which offers stabilization with low space and communication overhead, at the price of high stabilization time—$\Omega(\mathsf{diam})$ time units is a trivial lower bound, where diam denotes the network diameter. In this paper I survey some attempts I participated in, to study a different type of solutions we call *time-adaptive protocols* [22], or *fault local* in the original paper [23]: these are protocols which recover from a limited number of state-corrupting faults very quickly, typically at the cost of higher space and communication overhead. It is the belief of the author that this end of the tradeoff spectrum is worthy of further exploration.

in [23], and, later, in [22] we looked at one of the basic building blocks of time-adaptive protocols, called the *persistent bit* problem [23]. The task is to maintain the value of a bit in spite of state-corrupting faults. In some sense, the persistent bit problem captures the essence of the state-corrupting faults: maintaining a value is a trivial task if no faults are involved, but it is not a simple matter in face of state corruption. It seems that any interesting time-adaptive protocol embeds a solution to the persistent bit problem in it.

When speaking of time adaptiveity, we must say that some of the results show that we need to distinguish between different times. One corresponds to the concepts of *output stabilization* and the other corresponds to *state stabilization*: output stabilization is said to occur once the externally observable portion of the state ceases to change, and state-stabilization is said to occur when the internal state ceases to change as well. We describe proofs that these times often differ.

Results for Static Problems In [23], the persistent bit problem was introduced, where the goal is to retain the value of a common replicated bit across the system in spite of transient faults which may corrupt processors state arbitrarily, so long as the state at a majority of the processors is not faulty. The bit value is required to be equal at all processors, and the common value should persist across faults. An algorithm was presented with output stabilization time $O(f \log n)$ for $f = O(n/\log n)$. However, there the number of faults is cumulative: the algorithm cannot correct more than $O(n/\log n)$ faults throughout the execution of the system. In a certain sense, therefore, the state stabilization time of the algorithm of [23] is infinity.

The main positive result of [22] (Theorem 4.1 here), is an algorithm for the persistent bit problem. Let n denote the number of nodes in the system (we use the terms "nodes" and "processors" interchangeably), and let f denote the number of faulty processors in the start state. Let diam be the actual diameter of the network at the start state. The algorithm guarantees, for any $f < n/2$, that the output is recovered everywhere in $O(f)$ time units, and that complete state stabilization occurs in $O(\mathsf{diam})$ time. The algorithm is fairly robust in the sense that f and the network topology need not be known in advance. The algorithm can be simplified (and its space and communication requirements are reduced) if a smaller upper bound on f is known *a priori*.

However, the results in [22] fall short of the original goal. First, strictly speaking, the algorithm is not *self*-stabilizing, where the requirement is to recover from any number of faults $f \leq n$ (note, however, that preserving the value of a bit is somewhat meaningless when $f \geq n/2$); secondly, the algorithm is stated and proved correct only for synchronous networks; and thirdly, the algorithm has high space and communication complexity. Moreover, [?] treats only non-reactive tasks. Some of these shortcomings were corrected in later papers, mentioned in the sequel.

The negative results of [22] (Theorem ?? here), give lower bounds on the stabilization complexity of the persistent bit problem. The algorithm is optimal in terms of stabilization times: there is no algorithm for the persistent bit problem with output stabilization time $o(f)$, or state stabilization time $o(\mathsf{diam})$. We note that the lower bounds are proved for the synchronous executions model (and hence for the asynchronous model).

In [21], we announce that we can now prove the following strictly stronger result.

Theorem 1.1 *There exists a protocol for the persistent bit problem in the asynchronous network model such that if the state of $f \leq n$ of the nodes is corrupted, then the output bits stabilize everywhere in $O(f)$ time units, and complete state stabilization occurs in $O(\mathsf{diam})$ time units, where* diam *denotes the diameter of the network.*

The improvement is twofold: first, the new protocol works in the asynchronous network model; and second, it can withstand any number of faults (i.e., it is self-stabilizing).

Results for Dynamic Problems In [11], we present solutions for the reactive problem that is often used as a benchmark for such protocols: the problem of *token passing*. We treat the realistic case, where no bound is known on the time a node can hold the token (a node holds the token as long as the node has not completed some external task). We study the scenario where up to k (for a given k) faults hit nodes in a *reactive asynchronous* distributed system by corrupting their state undetectably. The exact number of faults, the specific faulty nodes, and the time the faults hit are *not* known.

We present several algorithms that stabilize into a legitimate configuration (in which exactly one node has a token) in time that depends only on k, and not on n (the number of nodes). The solutions are presented in stages. First, a basic protocol is shown to stabilize in $O(k^2)$ time and use only a constant number of (logarithmic size) variables per node. For this first protocol it is required that k is smaller than $\sqrt{n}$, that is, the first protocol does not self-stabilize (for large k). In terms of the number of individual nodes' steps the stabilization takes $O(kn)$ steps, and it is shown that any 1-stabilizing algorithm (that is, when $k = 1$) must use at least $n - 3$ steps.

The other algorithms are built on the basic one: one stabilizes in $O(k^2)$ time and is self-stabilizing (so k can be larger than $\sqrt{n}$), another enhanced version stabilizes in $O(k)$ time (and is time opti-

mal) but the space it uses is larger by multiplicative factor of k.

In [25] we outline the first proof that all reactive specifications admit adaptive protocols. In the reactive system model [28], 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. (Both the inputs and the outputs are distributed: an output of a node may depend on values input at various remote nodes at various times.)

The key ingredient of the proof is a new algorithm for distributing input values in an adaptive stabilizing fashion. Our algorithm is optimal, up to a constant factor, in the following measures:

- *Response time*: the time it takes an input value to influence the relevant output values, starting from the time of injection.
- *Recovery time*, which consists of two measures [18, 22]: *Output stabilization* is the time it takes until the outputs exhibit correct values after a fault. *State stabilization* is the minimum time between faults, for which the system is still guaranteed to have this output stabilization time and this resilience.
- *Resilience*: the severity of faults from which the algorithm fully recovers.

While the time complexity measures are rather intuitive, the resilience measure is new and requires an explanation. Let us first motivate it with the following little example. Suppose that a value is input to a node at some time, and then immediately a fault corrupts the state of that node. Clearly, there is no way for the system to recover the input value in that case. More generally, if a value is input to a node v at time t, and at time $t_f \geq t$ all nodes in radius $t_f - t$ around v are hit by a fault, then, by the same reasoning, the input value may be irrecoverable lost. This consideration leads us to the following definition.

Definition 1.1 *A value input to the system at time t is said to be* (ρ, β)-established *at time t_1 with respect to a given fault, if less than a fraction β of all nodes in radius $\rho(t_1 - t)$ around the origin of the value are affected by the fault, for some $0 \leq \rho, \beta \leq 1$.*

If an algorithm can recover all (ρ, β)-established values, we say it is (ρ, β)-resilient. The parameter ρ is called the *agility* of the algorithm, and β is its *strength.* Trivially, both the agility and the strength are at most 1. For our algorithm, we have $\rho \approx 0.023$ and $\beta = 1/2$.

Strong Time Adaptivity In [24] we wanted to take advantage of the fact that various subtasks in a huge network interest only parts of the network, and it is desirable that those parts, if non-faulty, do not suffer from faults in other parts. Our approach is to refine the previously suggested notion of time adaptive algorithms (or, another name we used: fault local algorithms), that was best suited for global tasks, for which the complexity of recovering was proportional to the number of faults. We refine this notion by introducing the concept of *tight fault locality* to deal with problems whose complexity (in the absence of faults) is sub-linear in the size of the network. In the context of the other papers mentioned, this can be called also *strong time adaptivity.* For a problem whose time complexity on an n-node network is $T(n)$ (where possibly $T(n) = o(n)$), a tightly fault local algorithm recovers a legal global state in $O(T(x))$ time when the (unknown) number of faults is x.

This concept is illustrated by presenting a general transformation for MIS algorithms to make them tightly fault local. In particular, our transformation yields an $O(\log x)$ randomized mending algorithm and an $\exp(O(\sqrt{\log x}))$ deterministic mending algorithm for MIS. The methods used in the transformation may be of interest by themselves.

2 Related Work.

The study of self-stabilizing protocols was initiated by Dijkstra [12]. *Reset-based* approaches to self-stabilization are described in [20, 6, 9, 10, 15]. In reset-based stabilization, the state is constantly monitored; if an error is detected, a special reset protocol is invoked, to consistently establish a correct global state, from which the system can resume normal operation (either some agreed upon state, or [15] a state that is in some sense "close" to the faulty state.) One of the

main drawbacks of this approach is that the detection mechanism triggers a system-wide reset in the face of the slightest inconsistency.

The distinction between output stabilization and state stabilization has been used and discussed in a number of papers. For example, in [8] it is noted that the output stabilizes in $O(\mathsf{diam})$ time, while the state stabilization time may be much larger. Parlati and Yung [31] and Dolev *et al.* [17] study a few cases where state stabilization coincides with output stabilization. Ghosh *et al.* [18] explicitly distinguish between output and state stabilization ('fault gap'). In [15], a distinction is made between a state-corrupting fault which triggers a reset, and a topological change which results in a milder effect.

The papers most closely related to our work are [18, 4]. In [18], an algorithm for the following problem is presented: given a self-stabilizing non-reactive protocol, produce another version of that protocol which is self-stabilizing, but whose output stabilization time is $O(1)$ if $f = 1$. The transformed protocol has $O(T \cdot \mathsf{diam})$ state stabilization time, where T is the stabilization time of the original protocol (no analysis is provided for output stabilization time when $f > 1$). The protocol of [18] is asynchronous, and its space overhead is $O(1)$ per link. However, it requires a self-stabilizing protocol to start with, and it may suffer a performance penalty in the case of $f > 1$. In [4] faults are stochastic, and consequently the correctness of information can be decided with any desired certainty less than 1. Under this assumption, a time-adaptive algorithm is presented. The algorithm handles both Input-Output relations, and reactive tasks. Additional examples for the special case of $O(1)$ recovery time appear in [13, 30, 15, 29]. For adaptive reactive protocols, Afek and Dolev [4] give an adaptive algorithm for the self-detection case, where each node can find on its own whether it is faulty.

3 Model

The system topology is represented by an undirected connected graph $G = (V, E)$, where nodes represent processors 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 $\mathsf{dist}(u, v)$. 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 $\mathcal{N}(v) = \mathsf{ball}_v(1) - \{s\}$: the *neighbors* of i.

A *distributed protocol* is a specification of the space of *local states* for each node and a description of the *actions* which modify the local states. As a part of each local state, there are distinguished *input* and *output registers* visible to the *external environment.* The environment can take two types of actions: input injection, i.e., assigning values to input registers, or fault injection, i.e., arbitrarily changing the state of an arbitrary set of nodes. The nodes whose state was 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. We say that the faulty nodes *underwent a fault* at time t_f.

In most papers (but not in all) we assume that the system is synchronous. For simplicity, we abstract the underlying communication mechanism by assuming that actions may depend only on the local state and the state of adjacent nodes. (It is known how to translate this model to the message-passing model; see, e.g., [6, 16].) An *execution* of the system is a sequence of synchronous steps: at 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.

4 The Persistent Bit Problem

The problem. The Persistent Bit problem is defined as follows. Each node maintains an externally observable *output bit* which satisfies the following conditions (this is a degenerate case of input/output relation: there is no input).

- *Eventual Agreement.* All output bits must be equal, except perhaps for a finite time, called *output stabilization time* immediately following a fault.
- *Persistence.* If the number of faults f in a given start state satisfies $f < n/2$, then the eventual common value of the output bits is equal to the common value of the majority of the nodes in the start state.

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

Our goal is to find a protocol with the smallest possible output and state stabilization time. We proved the following result:

Theorem 4.1 *There exists a protocol for the persistent bit problem such that if the state of $f < n/2$ of the nodes is changed arbitrarily, then the output bits are restored everywhere in $O(f)$ time units, and complete state stabilization occurs in $O(\mathsf{diam})$ time units, where* diam *denotes the diameter of the network.*

We proved that output- and state-stabilization times above are the best possible. However, we also discussed a simplified (and more efficient) solution under the assumption that a better upper bound on f is known.

Overview of the protocol The main difficulty in a time-adaptive solution to the persistent bit problem is that output values are required to change quickly, while old information must not be deleted too early. To see that, consider a given network, a node i_0 in it, and let N_0 be the set of all nodes at distance t or less from i_0. Suppose that node i_0 starts with output bit 0, and all the nodes in N_0 start with output bit 1. Clearly, in the first t time units, i_0 cannot distinguish between the cases of (a) all nodes in the system except i_0 have output bit 1, and (b) only the nodes in N_0 have output bit 1 but all other nodes have output value 0.

By the problem specification, the output value at node i_0 should be different in each of these cases. However, at the first t steps i_0 must deem itself faulty, because of the possibility of case (a) above. Moreover, if the protocol is fault-local, the output stabilization time is $O(f)$ for any f, and i_0 is forced to change its output value to 1 after $O(1)$ steps, and keep it 1 at least through time t, in line with case (a). However, it may later turn out that case (b) is true. Thus, if i_0 removes all traces of its previous output value in the first t steps, i becomes, in effect, an additional faulty node, which might adversely affect other non-faulty nodes later.[1] The intuitive conclusion is that nodes should not purge their old state even when they flip the value of their output bit. On the other hand, if the original value is never forgotten, then additional faults have cumulative effect, rendering the solution non-stabilizing. A satisfactory solution to the Persistent Bit problem, which is both fault-local and stabilizing, needs therefore hit a delicate balance between keeping old information and modifying it.

Our solution consists of two parts: one responsible for speedy stabilization of the output (while keeping old information), and the other for prudent state stabilization (removing obsolete information). Each node has, in addition to the externally visible output bit, another bit we call 'input bit.' Intuitively, the input bit serves as a long term memory in the sense that it is used infrequently to save the current value of the frequently-changing output bit. In a legal state, all these bits (in all nodes) are equal. The main component of the output stabilization part is the *regulated broadcast* protocol, which ensures that in $O(f)$ time, each node knows the true value of the input bits of sufficiently many nodes, and no node has a wrong estimate of any input value of any other node. The output bit is computed locally by a simple majority rule over these estimates. The key to state stabilization is the *input fixing* protocol, which guarantees that all faulty input bits are corrected in $O(\mathsf{diam})$ time units, while making sure that input values at non-faulty nodes never change. In the remainder of this section, we give some details regarding the first task.

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

[1] Interestingly, in [27] it is shown that there are graphs where a subset of $\Theta(n^{2/3}\mathsf{diam}^{1/3})$ nodes are the majority in all neighborhoods (up to distance about $\frac{\mathsf{diam}}{2}$) for all nodes in the graph.

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

The RB protocol avoids this problem. (It is very similar to the *power supply* technique, suggested independently in [2].) The protocol builds a tree rooted at each input value, and uses flooding to forward the root value of that tree. However, this 'broadcast wave' is slowed down to half speed (this is done by exposing the internal value only after an additional copy step). At the same time, nodes keep verifying the integrity of the tree and the broadcast information; if an inconsistency is found, a 'reset wave' is initiated; this wave progresses at full speed down the broadcast tree, erasing all estimates and tree structure as it goes (but does not harm the other trees). Because of the speed difference, a wrong value cannot reach too far before it is eliminated.

Lower Bounds on Stabilization Times We proved that the output and state stabilization times of the algorithm of [22] for the persistent bit protocol are asymptotically optimal.

Saving Space One way to reduce the space complexity is to assume that there is a known upper bound F on the number of processors corrupted in a single transient fault. Under this assumption, the algorithm can be converted to one with output stabilization time $O(f)$, state-stabilization time $O(F)$, and space complexity $O(F)$.

5 The Asynchronous and Self Stabilizing Algorithm of [21]

The high-level structure of the protocol of [21] is identical to that of the algorithm presented in [22]

The internal structure of each of the modules is different, though. Let us first describe the change which allows the protocol to be fully self-stabilizing in the synchronous model. For that purpose, only a change in the input fixing protocol is needed, while the output stabilization part remains unchanged. The key is the fact (established in [22]) that if the input bits do not change their values for $2 \cdot \mathsf{diam}$ consecutive time units, then all nodes agree on the values of all input bits (since these values are broadcast at the rate of two pulses per link). We therefore change the input fixing algorithm as follows. In the old input-fixing algorithm, each node estimates diam, and counts up to $2 \cdot \mathsf{diam} + 1$; when the counter hits the bound, the input bit is set to the value of the output bit. In the new algorithm, we force all counters to have the same value and to count up to $6 \cdot \mathsf{diam} + 1$. Forcing all counters to be equal is done by taking the counter value, at each step, to be one plus the minimum of all neighbors' counters (including the local counter). With this modification, it is easy to prove the following lemma.

Lemma 5.1 *If some input value changes its value in the time interval* $[0, 2 \cdot \mathsf{diam}]$*, then no input value is changed in the time interval* $[4 \cdot \mathsf{diam}, 6 \cdot \mathsf{diam}]$.

Lemma 5.1 guarantees that in any case, an interval of $2 \cdot \mathsf{diam}$ time units without any change in the input values will occur, and with this it is simple to show that complete state stabilization occurs in $O(\mathsf{diam})$ time units regardless of the start state.

To make the algorithm asynchronous, we need to make a deeper change, both in the input fixing and in the output stabilization protocols. These changes are based on combining known techniques with some new ideas. For the output stabilization part, we observe that the "power supply" algorithm by Afek and Bremler [2] can be slightly modified to have the properties needed for the persistent bit problem. The idea is to synchronize each broadcast tree by a stream of "ticks" generated by the root. The effect of slowing down the broadcast but not the reset is obtained by having the broadcast, in every ho wait untile it "consumes" two ticks. More details can be found in [2].

The input fixing protocol has two parts: the depth computation and the counter synchronization, and we need to modify both. For the depth

computation, we have developed a little protocol based on the Bellman-Ford Algorithm which never underestimates the depth, and stabilizes in $O(\mathsf{diam})$ to the correct value. For the synchonisation part, we use a variant of the self-stabilizing synchronizer described in [8]. We note that the fact that the stabilization time of that synchronizer is $O(\mathsf{diam})$ presents no problem since state stabilization cannot occur in $o(\mathsf{diam})$ time units anyway (see [22], Theorem 4.3). Finally, we combine the synchronizer pulses with the regulated broadcast ticks.

6 Reactive Tasks: the Case of Mutual Exclusion

In this problem it is required that exactly one node "possesses the token" (i.e. a locally computable predicate $TOKEN$ holds for that node) at any moment, and that every node eventually holds the token.

Note that this is a reactive system: a node P holds the token until some outside entity signals P, and P alone, that P can release the token.[2]

Let *k-stabilizing* protocols be time adaptive protocols for the case that an upper bound $k \leq n$, on the number of faults, is known (where n is the total number of processes). That is, k-stabilizing protocol stabilizes in a time proportional to k and not to n. Some of the protocols in [11] are k-srabilizing, while others are time adaptive even if k is not known. The latter, however, consume more space. (Note that in [22] knowledge regarding the number of faults saves space.)

New Techniques: Let us first mention the techniques used here, that we hope will be proven useful also for other reactive tasks as well. For that consider the *propagation of inputs* and the *propagation of faults*: Intuitively, in a non-trivial reactive system, nodes change their states as a result of the states of their neighbors; for example, when a node P stops holding the token, and its neighbor P' starts holding the token as a result. This *propagates* (to P') the input that told P to release the token. If, however, P acted as a result of a fault, then P' should not have changed its state; now that it did, P's state is now corrupted, and we say that the fault has *propagated* (to P). Intuitively, the techniques we use here bound the propagation of faults. Such bounding is essential for time adaptivity, since, if faults propagate to the whole network, any recovery process would have to be global too. Techniques for bounding were shown in the papers mentioned above for non-reactive systems. However, bounding is more difficult in reactive systems, since such systems must still propagate the inputs.

Results: In that paper we use the two common time related complexity measures (See the definition in [11]) (1) The sum (over all nodes i) of *steps*, where in one (atomic) step, node P reads a neighbor's state, computes, and writes P's variables. (2) *asynchronous time*, or *rounds*, the time assuming (for the sake of time complexity calculation only) that no step lasts longer than one time unit, and that nodes take steps in parallel.

We present two k-stabilizing algorithms for token passing over an asynchronous ring of nodes (processes). The first stabilizes in $O(k^2)$ rounds, or $O(k^2 n)$ steps, using a constant number of variables per node (each of $O(\log n)$ bits).

The second algorithm can be viewed as a "parallelized" version of the first. Its round complexity is $O(k)$. Thus it is (asynchronous) time optimal. However, the space it uses is larger by a multiplicative factor of k.

On the negative side we show that any token passing self stabilizing algorithm requires at least $\Omega(n)$ steps, and thus its step complexity cannot be a function of k alone. (This also means that our algorithms are step optimal for a constant k.)

7 General Adaptive Tasks, [25]

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 $0 \leq t' \leq t$. A

[2] (An alternative assumption, that other nodes are aware of the signal (e.g. if it arrives after a fixed time), would have simplified the task considerably; However, this would not have been a realistic assumption, since, in reality, the signal models the completion of some local (mutual exclusion user) task at P.)

reactive problem is specified by a function mapping each time step to a binary relation over the input and output assignment. 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 relation 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 Adaptive Bit Distribution problem (SABD). Each node v has an *output bit* denoted $\mathtt{out}_v$. A special node called *source*, denoted by s, gets an *input bit* $\mathtt{B}_s(0)$ from the environment at time 0. The goal is that eventually, $\mathtt{out}_v$ holds $\mathtt{B}_s(0)$. The difficulty is that at some unknown time $t_f \geq 0$, the state of a subset $F \subseteq V$ of the nodes is corrupted arbitrarily, for an unknown $|F| = f$.

The Concept of core Because of the lack of space, we explain here only the intuition behind the main new technique used by the algorithm. At time 0, the source gets the input value, it assigns it to $\mathtt{B}_s$ and starts a regulated broadcast. The idea is to quickly, but carefully, create replicas of the original input bit. Thus, if faults hit only a small number of nodes, most of the replicas remain correct. Moreover, we should be quick, so that the later the faults occur, the more correct replicas there are, and the larger is the resiliency of the algorithm. On the other hand, we should be careful not to replicate a value that was already corrupted by faults.

The local replica at a node v is called $\mathtt{B}_v$. For any time $t > 0$, let $\mathsf{heard}(t)$ denote the set of nodes that were reached by the broadcast (e.g., $\mathsf{heard}(0) = \{s\}$). To better understand the difficulty, note that a state corrupting fault may change the values of the B variables, including the value of $\mathtt{B}_s$. Thus, even a non-faulty node joining $\mathsf{heard}(t)$ by receiving an RB may, in fact, be receiving an incorrect value for B. The central idea of the algorithm is to let such a joining node use, and help maintain , the following invariant.

Definition 7.1 *The* core Invariant: *At each time step t there exists a set of nodes* $\mathsf{core}(t)$ *such that the majority of the values of* $\{\mathtt{B}_u(t) \mid u \in \mathsf{core}(t)\}$ *are equal to the input bit.*

Assume for now that the invariant holds true. If a node v could consult all the nodes in $\mathsf{core}(t)$ before committing to a value for $\mathtt{B}_v$ at time t, then the value assigned to $\mathtt{B}_v$ would be correct as well. This approach raises two questions: first, how is $\mathsf{core}(t)$ to be defined, and second, how can v find the values of the $\mathtt{B}_u$ variables for nodes in $\mathsf{core}(t)$ at time t. As for the second question, we shall see that regulated broadcast suffices, assuming that a fault may hit only a minority of $\mathsf{core}(t)$. (This is the "approximate" answer: Node v will, actually, be able to consult the authentic votes of $\mathsf{core}(t)$, only by some time $t' > t$.)

It remains to explain how can $\mathsf{core}(t)$ be defined constructively. One possible choice is:

Example 7.1 $\mathsf{core}(t) = \{s\}$ *for all t. With this definition, an algorithm would have had zero agility- the number of faults for which the algorithm is resilient does not grow with time. If the original* core *is corrupted (by just one fault), the* core *invariant is violated.*

For better fault resilience, it is desirable that core grows as fast as possible, while preserving the invariant. The first idea for maintaining the invariant for a growing core is inductive: a node v would join the core only after verifying that the value it assigns to $\mathtt{B}_v$ is the majority value of the current core. This improves the agility, but still does not lead to an optimal one. Consider the following example for a choice of $\mathsf{core}(t)$.

Example 7.2 *Let the* maximum core *at time t be all the nodes in* $\mathsf{heard}(t-1)$. *Intuitively, a node v, at distance d from the source s consults all the nodes the are closer to the source. Thus v has to collect a vote from a node u such that* $\mathsf{dist}(u,v) \approx 2d$. *Moreover, the earliest that u can join* heard *and* core *is time $d-1$. Thus, the radius of the* core *at any time t is $O(\sqrt{t})$. In terms of Def. 1.1, we have $\beta = 1/2$ but ρ is not a constant. Thus ρ here is not optimal. In our algorithm we manage to have constant ρ.*

We formalize the requirements of core as follows. (Example 7.2 motivates the notion of t^c (as opposed to t in Requirement 3.)

Requirements for core:

1. $\mathsf{core}(0) = \{s\}$.
2. For all $t \geq 0$, $\mathsf{core}(t) \subseteq \mathsf{core}(t+1)$.
3. If $v \in \mathsf{core}(t+1) - \mathsf{core}(t)$, then there exists a time $t^c \leq t$, such that v can verify (if v is non-faulty) by time t that all votes from nodes in $\mathsf{core}(t^c)$ are authentic.
4. Except for the case of $\mathsf{core}(0) = \{s\}$, the first core consulted by a node v, does not include v.

Note that in Requirement 3 Node v verifies the authenticity of votes of nodes in some $\mathsf{core}(t^c)$, as opposed to $\mathsf{core}(t)$. Intuitively, this relaxation in the requirement generates a trade-off in the value of ρ. Optimizing the trade-off enables us to improve ρ beyond that of Example 7.2.

We choose $\mathsf{core}(t) \stackrel{\text{def}}{=} \{v \in V \mid \mathsf{dist}(s,v) \leq \alpha t\}$, for a constant α we determine later in that paper.

References

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

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

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

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

[5] S. Aggarwal and S. Kutten. Time-Optimal Self-Stabilizing Spanning Tree Algorithms. In *Proc. of the Thirteen Conference on the Foundations of Software Technology and Theoretical Computer Science* (FSTTCS), Bombay, India, December 15-17, 1993.

[6] 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.*

[7] H. Attiya and J. Welch. *Distributed Algorithms.* McGraw-Hill Publishing Company, UK, 1998.

[8] 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).

[9] 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.

[10] 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.

[11] J. Beauquier, C. Genolini, and S. Kutten. Optimal reactive k-stabilization: the case of mutual exclusion. In *Proc. 18th Ann. ACM Symp. on Principles of Distributed Computing*, pages 209–218, May 1999.

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

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

[14] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

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

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

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

[18] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemamraju. Fault-containing self-stabilizing algorithms. In *Proc. 15th Ann. ACM Symp. on Principles of Distributed Computing*, May 1996.

[19] G. Itkis and L. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico*, pages 226–239, Nov. 1994.

[20] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th Ann. ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, Aug. 1990.

[21] Asynchronous Time-Adaptive Self Stabilization. S. Kutten and B. Patt-Shamir. A Brief Announcement in the *Proc. 17th Ann. ACM Symp. on Principles of Distributed Computing*, Puerto Vallerta, Mexico, June 1998.

[22] S. Kutten and B. Patt-Shamir. Time-adaptive self-stabilization. In *Proc. 16th Ann. ACM Symp. on Principles of Distributed Computing*, pages 149–158, Aug. 1997.

[23] S. Kutten and D. Peleg. Fault-local distributed mending. In *Proc. 14th Ann. ACM Symp. on Principles of Distributed Computing*, Aug. 1995.

[24] S. Kutten and D. Peleg. Tight fault locality (extended abstract). In *36th Annual Symposium on Foundations of Computer Science*, pages 704–713, 1995.

[25] S. Kutten and B. Patt-Shamir. Adaptive Stabilization of Reactive Distributed Protocols. Extended Abstract. http: iew3.technion.ac.il:8080/ kutten/kp00.ps.

[26] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Mateo, CA, 1995.

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

[28] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 3:609–678, 1993.

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

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

[31] 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.

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

[33] Gerard Tel. Distributed Algorithms. Cambridge University Press, 1994.