Output Stability Versus Time Till Output
(Extended Abstract)

Shay Kutten[1,⋆] and Toshimitsu Masuzawa[2]

[1] Dept. of Industrial Engineering & Management, The Technion, Haifa 32000, Israel
[2] Graduate School of Information Science and Technology, Osaka University, Japan

Abstract. Consider a network whose inputs change rapidly, or are subject to frequent faults. This is expected often to be the case in the foreseen huge sensor networks. Suppose, that an algorithm is required to output the majority value of the inputs. To address such networks, it is desirable to be able to stabilize the output fast, and to give guarantees on the outputs even before stabilization, even if additional changes occur.

We bound the *instability* of the outputs (the number of times the output changes) of majority consensus algorithms even before the final stabilization. We show that the instability can be traded off with their *time adaptvity* (how fast they are required to stabilize the output if f faults occurred). First, for the extreme point of the trade-off, we achieve instability that is optimal for the class of algorithms that are optimal in their output time adaptivity. This is done for various known versions of majority consensus problem. The optimal instability for this case is $\Omega(\log f)$ and is shown to be $O(\log f)$ for most versions and $O(\log n)$ in some cases. Previous such algorithms did not have such a guarantee on the behaviour of the output before its final stabilization (and their instability was $\Omega(n)$). We also explain how to adapt the results for other points in the trade off.

The output stabilization in previous algorithms was adaptive only if the faults ceased for $O(\mathbf{Diam})$ time. An additional result in this paper uses adaptations of some previous tools, as well as the new tools developed here for bounding the instability, in order to remove this limitation that is undesirable when changes are frequent.

1 Introduction

Consider an action that is to be taken according to some value measured by sensors composing a network. The measurements at some of the sensors may be different (possibly, because of measurements inaccuracies, or because of faults). To overcome that, the network computes a majority consensus. An outside action may be taken according to this consensus. For example, travelers in the woods may consult the sensor near them to decide whether to unfold a tent, since a

⋆ This research was supported in part by a grant for research on sensor networks from the Israeli Ministry of Science and Technology.

A. Pelc (Ed.): DISC 2007, LNCS 4731, pp. 343–357, 2007.

storm is coming, or whether to fold the tent and continue the walk in the case that the rain has gone.

A fast answer is sometimes crucial. However, if one insists that a warning always come as fast as possible, then some false warnings are unavoidable. For example, if all the near-by sensors predict a storm, then the initial answer must be that a storm is about to break. If those near by sensors are a small minority, then the answer changes eventually. The false alarm may have consumed some resources (e.g., the effort of unfolding and then re-folding). An external system using the output may even act incorrectly if the output changes too frequently. For example, various machines, if turned on and off too often, would break.

Note, that changes in the output may be unavoidable even if the travelers are willing to wait. The inputs of the sensors may change, for example, when a chance of a storm is indeed increasing. Moreover, the inputs change not all at the same time. Hence, at some point, the travelers should decide that they are willing to act upon the best answer the network can give them at present.

The stability (or instability) of the output for a distributed consensus was discussed in [1,2]. It is the number of times the output may change when the input changes. The output time complexity, or the output stabilization time, in self stabilizing systems[1], is the time it takes for the system to start outputting the correct and final output, following input changes (e.g. introduced by faults). These two measures of complexity seem, intuitively, related. (For example, the number of changes in the output of a node cannot be larger than the time complexity). Still, they were discussed only separately in the literature. The stability problem was studied also outside the setting of distributed systems, in the context of mechanical engineering, see e.g. [11].

Note, that instability cannot be avoided altogether. In every setting of the Consensus problem, some inputs dictate a certain output value, while some others dictate a different one. For any other set of inputs, the algorithm enjoys the freedom to decide the value to output. In [1], they explore how the stability is increased (the instability is decreased) as a function of this freedom.

In this paper, we explore the way the freedom along another dimension influences the stability. In the Majority Consensus problem, the output value must be that of the majority (as opposed to cases studied in [1]). However, (a different kind of) freedom exists in this system too: if there are changes in the inputs of some f nodes (or any other changes by f state faults), then the output is permitted to be incorrect (i.e., different than the majority of the inputs) for some finite time. We note that allowing such a freedom is unavoidable in distributed systems where inputs may change. This is because it takes some time for a changed input in one node to be communicated to the other nodes. Such a freedom is especially assumed in the context of self stabilization [18] (except for the case of a very limited set of tasks [8] that does not include the task of Consensus).

In optimal *time adaptive* systems [16,12], this unavoidable output freedom is restricted to the unavoidable duration, as a function of the number of faults. For

[1] Self stabilization and stability are two different notions. The use of both may be confusing, but we chose not to change terms that are common in the literature.

Majority Consensus, if f faults occur, the output is required to *stabilize* to the correct final value in $O(f)$ time.

Definition 1. *Assume, that the outputs of all the network nodes is* stable *(will not change unless the inputs are changed) at some time t_0. Assume further, that a set of faults and changes occur at some time $t_f > t_0$. The* instability *of an algorithm is the number of times the output will change in the worst case starting from t_f and until the output eventually stabilizes (if it ever does) or until further faults or input changes occur.*

Several time adaptive algorithms exist for the Persistent Value Problem [16,12] and the Majority Consensus with Persistence Problem [21]. For these algorithms, the instability for an optimal time adaptive algorithm was $\Omega(n)$. This means that the output of a node could change every time unit until the final stabilization. Some algorithms give (different kinds of) guarantees for outputs even before the final stabilization, but these guarantees apply only for non- faulty nodes, and the algorithms were not time adaptive. See, e.g. [15]. Self stabilizing algorithms in general have been criticized for not giving much guarantee for the value of the outputs before stabilization. This is especially problematic in networks that rarely stabilize. Hence, reducing the instability (especially of faulty nodes) below $O(n)$ can be viewed as a step in the right direction for such network. Another result the can be obtained using our methods for and with fast changing networks as a motivation, is time adaptivity even when additional faults occur before stabilization.

Main results: We address the problems of (One Time) Majority Consensus, Persistent Value (and the related "Majority Consensus with Persistence"), and Repeated Majority Consensus. On the negative side, it is easy to show that no algorithm for these problems that is asymptotically optimal in its time adaptivity can have instability that is better than $\Omega(\log f)$. This is the case even if the algorithm is not required to self stabilize. We then present algorithms that both have optimal time adaptivity and have $O(\log f)$ instability for non-faulty nodes, and $O(\log f)$ (for some cases) or $O(\log n)$ (for other cases) for faulty nodes. That is, our algorithms are asymptotically optimal in their output stability for the class of optimal time adaptive algorithms. We then show how to generalize the results, such that if the time complexity is allowed to grow beyond $O(f)$, the instability shrinks below $\Omega(\log f)$. The instability of our algorithms in most cases is adaptive too, that is, it does not depend on n, but only on f.

Additional results: The proofs of the following additional results are deferred because of space considerations, and do not appear in this extended abstract. While previous algorithms for the Persistent Value Problem self stabilized in any case (as do the algorithms in the current paper), they were time adaptive only when the faults occurred in one batch at a time, and no additional faults occurred until the eventual full state stabilization (not just until output stabilization). This happens in $\Omega(n)$ time. Our results can be proved for a more realistic model, where faults can occur at any time, and not just in batches.

As tools for the our algorithms, we had to design a building block broadcast module (Protocol ABC) that is both error confined ([15]) and time adaptive ([12]). In contrast, the error confined tool of [15] was not time adaptive, while the time adaptive tool of [12,19] was not error confined. We also had to add to the tool a snap stabilizing (see [8]) action we term `Cancel`.

2 Model, Definitions, and Some Very Related Work

The system is modeled as a fixed undirected connected graph $G = (V, E)$, where $|V| = n$. Nodes represent processors and edges represent bi-directional communication links. Every node has a unique identity `ID` that cannot be changed by faults. For the sake of this extended abstract, we assume that the network is synchronous, even though methods to translate protocols such as ours to asynchronous networks are known, such as in [19,21]. The distance between two nodes $u, v \in V$, denoted $\mathbf{dist}(u, v)$, is the minimum number of edges in a path connecting them. Given a node $v \in V$, let $\mathtt{Ball}_v(r) = \{u \in V \mid \mathbf{dist}(v, u) \leq r\}$ be the *ball* of radius r around v. The radius of the network around a node v, denoted $\mathbf{Radius}_v$, is the minimal r such that $\mathtt{Ball}_v(r) = V$. $\mathbf{RRadius}_v$ is the first power of 2 larger than or equal to $\mathbf{Radius}_v$. $\mathbf{Diam}$ (the diameter) is the maximum over all $v \in V$ of $\mathbf{Radius}_v$. For the purpose of saving in memory only, we assume that the diamter is bounded by $Max\mathbf{Diam}$. For simplicity of exposition, we assume often that $\mathbf{RRadius}_v = \mathbf{Radius}_v$. In the extended abstract, we assume that the topology of the network is known (this assumption is lifted in the full paper, see a short discussion in Section 6 below).

Definition 2. *A* state corrupting fault *is an action that alters the state arbitrarily in some subset of nodes. We terms these node* faulty*.*

For simplicity, we assume that after a fault, each node is in a legal *local* state (otherwise, the node can detect the fault). Our approach in modeling faults is similar to the one in [9,22]. The model of state-corrupting faults is implicit in the work of Dijkstra about *self stabilization* [18]: put in our terminology, a system is called *self-stabilizing* if after some arbitrary state-corrupting faults occur (possibly, hitting all nodes) the system starts behaving correctly eventually. (Issues arising from different definitions of self stabilization are discussed in [10]). See the full paper for more detailed definitions.

We find it convenient to define two sets $\Pi_{state}, \Pi_{output}$ of correct behaviors. For defining Π_{output}, we assume that every node has a part of its state called the *output*. Moreover, only assignment to this variable are external actions of the protocol. Hence, only such actions show in *behaviors* in this case. The specific legal behavior Π_{output} is given in the definition of every problem to be solved. We say that Π_{output} determines *output stabilization*.

A protocol is *time adaptive* for the output stabilization if the system starts behaving correctly after a time which depends only on f, the number of faults (rather than on n). In other words, the output stabilization time is $O(g(f))$ for some function $g(f)$. The problems we solve is defined next.

Definition 3. *The* One Time Majority Consensus *problem: Every node has an* input *that can be changed only once and only by the environment and an* output *variable read by the environment. (In the extended abstract, we assume that b is binary). For every node, it is required that,*
Eventual Agreement- *the output stabilizes to the majority value of the inputs eventually.*

To unify the discussion, we consider the majority value as the correct one, and nodes with the minority input as faulty. A second problem- the Persistent Value Problem, is defined below.

Definition 4. *The* Persistent Value problem *[16,12] (somewhat rephrased): A value b was given* exactly once *by the environment to every node (the same b to all the nodes) before the faults started. (In the extended abstract, we assume that b is binary). This value was stored in each node in a storage variable. To be compatible with previous papers dealing with the persistent value problem, we name this storage variable the* input *variable. This (and every other) variable can be changed by faults, or by the node. Every node also has an* output *variable read by the environment. It is required that:*
Persistence: *it is required that both the output and the input of every node stabilize to b eventually.*

We note that persistence implies eventual agreement. The algorithm we present here solves only the requirements for the output, while the requirement for the *input* variable is solved by the Input Correction Module taken from [12] (the current algorithm and the above module are executed as co-routines). We note that we termed the storage variable the *input* since that storage variable is the input for the algorithm module designed in the current paper (though it may be changed by the other module, the one taken from [12]). The properties of the Input Correction Module are listed in Subsection 5.2.

Note, that designing the Output Stabilization Module for the Persistent Value problem is a harder task than designing the One Time Majority Consensus, since our algorithm must take into account the fact that its input may be changed not only by the environment (at some time t_f the faults occur), but also (later) by the Input Correction Module.

In *Repeated Majority Consensus*, changes may continue to occur. The time and the number of changes are counted from the last time the network was stable.

Additional very related work: The distinction between output stabilization and state stabilization is used and discussed in a number of papers [4,16,12,28,3,5]. Fast stabilization of output variables has been demonstrated in a number of algorithms [7,6,24,25,16,17,26,19,5] and some general methods to achieve time adaptivity [12,26]. In [20], the importance of output stability for practical Internet protocols was emphasized and obtained using time adaptivity methods.

3 Lower Bound

We establish a lower bound for the instability of a protocol given that the output of the protocol is required to stabilize (to the correct value) as fast as possible (asymptotically). The requirement about the fast output stabilization is used heavily in the proof. Furthermore, we see in this section that relaxing this requirement leads to weakening the lower bounds. This establishes the lower bound side of the trade off between output time and instability. We note that the lower bound holds even for synchronous networks, and even for algorithms that are not required to self stabilize.

Theorem 1. *The instability of any deterministic asymptotically optimal time adaptive protocol for the Persistent Value Problem, or for the Majority Consensus Problem, is* $\Omega(\log f)$.

The formal proof are deferred to the full paper. Informally, it considers a line network with v as the first node in the line. In the first, say, X time units, v receives only *votes* (broadcasts of input values) from the nearest X nodes. If all of them vote some value b_1, then v cannot distinguish between this case and the case that $f = 0$. To be time adaptive, v must start outputting b_1 within some constant time C. (We then choose $X = C$). Next, assume that out of the C^2 closest to v, the majority ($C^2 - C$) vote b_0. Now, v must change its output to b_0 to be time adaptive (since it may be the case that $f = C$). Next, we consider the C^3 nodes closest to v. This argument is carried forward to show $\Omega(\log f)$.

A problem with initial algorithmic ideas: At first glance, the proof of the lower bound seems to suggest an algorithm: (1) collect *votes* (broadcasted input values of the other nodes), (2) after changing the output, do not change the output again, before the number of votes received is grows by a factor of some C. This would have implied a logarithmic instability in the case that no vote may change or may be corrupted by a fault. Unfortunately, it is possible that votes arriving at v (by broadcasts) cease to arrive, or change their value. This can be caused by faults at the sources of such votes, or at nodes on the route from them to v, or by the action of the algorithm that corrects the faults.

4 A Building Block: Error-Confined and Adaptive Broadcast

As mentioned at the end of Section 3, changes in a vote received at v increase the instability. Some such changes cannot be avoided, since they represent real changes in the inputs. We use a tool that avoids *some* changes- informally, it allows a node s to broadcast its input such that if s and a recipient v are not faulty, no node on the way from s to v can change the value received at v (though the protocol may fail to deliver any value sometimes). This is termed an *error confined* broadcast in [15]. See the exact properties of this protocol, ABC (*Adaptive Broadcast with Confinement*, in Theorem 2 (the definitions of the broadcast task and of error confinement appear in the appendix).

As compared in "Additional Results" above, these properties of ABC are a combination of those of the tools used in previous papers [16,12,21,8]. Still, if the algorithms of those papers are modified to use our tool, instead of their own tools, their instabilities would still be high. However, the ABC tool proved useful for our algorithms and may prove useful by itself in the future.

Definition 5. *The value of the broadcast of a node s received in a node v is* authentic *if it was indeed communicated by s (rather than a value resulting from a corruption in some channel or some intermediate node)* $2\mathbf{dist}(v, s)$ *time earlier. Operation* `Cancel` *performed by v on the ABC of s causes the value of s received at v to become undefined ($\perp$). Moreover, if v starts again receiving s's broadcast, then the value received is authentic, and was broadcast by s after the last* `Cancel` *of v (unless v itself suffered another fault meanwhile). (The motivation for this operation is similar to that of* snap stabilization. *[8]).*

The detailed description of protocol ABC and the proof of the following theorem are deferred to the full paper. They do not use cryptographic assumptions (but alternative implementations that do use cryptographic assumptions may save in communication complexity).

Theorem 2. *Consider any node v. Let t_f be the last time the faults occurred. Let t_b be the time that s started broadcasting a value b. Finally, let $t_{\texttt{Cancel}}$ be either the last time v performed* `Cancel` *on the broadcast of s, or the last time that the value of the broadcast of s at v became $\perp$ (whichever came later). Below, if some of these times t_i is undefined, then $\max\{t_i, t_j, t_k\} = \max\{t_j, t_k\}$.*

1. Speed: *As long as the value b broadcast by the source node s does not change, Protocol ABC of s at v outputs b starting at time $\max\{t_f, t_b, t_{\texttt{Cancel}}\} + 2\mathbf{dist}(s, v)$.*
2. Time adaptivity: *(even for faulty nodes): At any time t such that $t \geq \max\{t_f + f, t_{\texttt{Cancel}}\}$, the output value (for s's broadcast) is either authentic or is undefined (equal to $\perp$).*
3. Error Confinement *(for non-faulty nodes): Let $t_{f(v)}$ be the last time that a fault hit node v. Assume, that the vote of some s changed in v* **from** *some $b \neq \perp$ after time $t_{f(v)}$. Then, it first changes to $\perp$. In addition, starting from that time, the output value (for s's broadcast in v) is either authentic or is $\perp$. Finally, if the value does become authentic (after changing first to $\perp$ then is stays authentic, unless additional faults occur.*

5 Instability Upper Bounds

Theorem 3. *There exists a self stabilizing protocol for Majority Consensus, such that if the local states of f of the nodes are changed arbitrarily, then*

- *Time Adaptivity: The output values are restored everywhere in $O(f)$ time;*
- *Instability: The instability in each process is $O(\log f)$.*

Theorem 4. *There exists a protocol for the Persistent Value Problem such that if the local states of $f < n/2$ of the nodes are changed arbitrarily, then Time Adaptivity is achieved, and, in addition,*

- *Instability: The instability is $O(\log f)$ for non- faulty nodes and $O(\log n)$, for faulty nodes .*
- *Complete state stabilization occurs in $O(\mathbf{Diam})$ time units.*
- *There is no change in the input of any correct process.*

Note, that the requirement that $f < n/2$ in the statement of Theorem 4 is required by the Input Correction Module of [12], to ensure persistence (not stability, nor stabilization). We note that these are the best possible output- and state-stabilization times even when the instability is allowed to be higher [12]. However, if the time is allowed to be higher then the instability can be smaller. The protocols claimed in the theorems are presented in two subsections below.

5.1 Stable Adaptive Majority Algorithm (Theorem 3)

The algorithm for this easier problem is given in figure 1. Its informal description is given in the full paper. In short, each node broadcasts (using ABC) its input and collects the values broadcasted by the others. The node outputs the majority of the votes it receives, but only from a ball of radius Scanned around

```
Broadcast (using algorithm ABC) $\texttt{input}_v$;
Receive every arriving broadcast of other nodes;          (* possibly, $\perp$ *)

Let $\mathsf{NearestUndef} = \min\{\mathbf{dist}(s,v) \mid \texttt{value}_v[s] = \perp\}$;
                                   (* $\texttt{value}_v[s]$ is $s$'s vote as received in $v$ *)
If $\mathsf{NearestUndef} \neq$ Undefined then                  (* Received some $\perp$ *)
    Let $\mathsf{Reduced} \leftarrow \max\{\text{integer } i | 2^i < \mathsf{NearestUndef}\}$.   (* Reduce to exclude $\perp$ *)
    Set $\mathsf{Majority}$ to the majority value in $\texttt{Ball}_v(2^{\mathsf{Scanned}})$;
    $\mathsf{Suspects} \leftarrow \{w \in \texttt{Ball}_v(2^{\mathsf{Scanned}}) \mid \perp \neq (\texttt{value}_v[w] \neq \mathsf{Majority}$
    $\mathsf{Wait} \leftarrow \min\{\mathsf{Wait} - 1, |\mathsf{Suspects}|\}$;                (* Delay reducing Scanned *)
    If $\mathsf{Wait} \leq 0$ and $\mathsf{Reduced} < \mathsf{Scanned}$ then      (* without violating adaptivity. *)
        $\mathsf{Scanned} \leftarrow \min\{\mathsf{Reduced}, \mathsf{Scanned}\}$;
        $\texttt{Cancel}$ the broadcast of every node outside $\texttt{Ball}_v(\mathsf{Scanned})$;
        $\mathsf{Wait} \leftarrow |\texttt{Ball}_v(2^{\mathsf{Scanned}})|$.

If for every node $u$ in $\texttt{Ball}_v(2^{\mathsf{Scanned}+1})$ the arriving $\texttt{value}_v[s]$ is not $\perp$ then
    $\mathsf{Scanned} \leftarrow \min\{\log \mathbf{RRadius}_v, \mathsf{Scanned} + 1\}$;
    $\mathsf{Wait} \leftarrow |\texttt{Ball}_v(2^{\mathsf{Scanned}})|$;
    $\texttt{Cancel}$ the broadcast of every node outside $\texttt{Ball}_v(2^{\mathsf{Scanned}})$.

Set $\texttt{output}$ to the majority value in $\texttt{Ball}_v(2^{\mathsf{Scanned}})$.
```

Fig. 1. Stable Adaptive Majority Consensus Algorithm (with adaptive instability)

it. The algorithm decreases Scanned, carefully, to exclude votes it suspects their authenticity, and increases Scanned, carefully, when it guesses that votes in a larger ball are authentic. It is easy to demonstrate that changing the algorithm so that it would change Scanned more "drastically" (e.g. would restart from Scanned $= 0$ every time a vote changes) would either not be adaptive, or would have high instability, or both. For example, the event that some vote becomes $\perp$ can happen f times. Had the algorithms restarted to Scanned $= 0$ each time some vote became $\perp$, the instability could have grown to $\Omega(f)$.

Had there been only increases (the last "If" statement), it seems easy to prove the $O(\log n)$ upper bound (maybe also the $O(\log f)$). Bounding the number of decreases in the radius (the first "If" statement) is somewhat harder. Still harder, is bounding the number of decreases to be $O(\log f)$ rather than $O(\log n)$ even in a node v that is faulty. Intuitively, all the votes that a faulty node v "believes" it received, it may have not received actually (which means that they are not authentic in v), so the effect on the instability at v is as if there were n faults. The "wait" mechanism in the code is intended to allow non- authentic votes at v to disappear before decreasing Scanned. The node cannot wait too much, however, since this would have caused it not to be time adaptive. Hence, the node waits as much as possible given a lower bound (|Suspects|) it computes for the number of faults. This Wait method bounds the number of times Scanned is decreased, even in a faulty node.

Lemma 1. *The instability of the Stable Adaptive Majority Algorithm of Figure 1 is* $O(\log f)$.

The proof of the lemma is deferred to the full paper. We bring here only the most interesting case (that uses the Wait mechanism). This is the case that the value of Scanned is reduced at some time $\tau_0 < Cf$. The main reason this case is interesting, since this is the case that votes in v may not be authentic (they become either authentic or $\perp$ after $O(f)$ time). This can have an effect on v that is similar to a number of faults that is larger than f, which makes the proof of $O(\log f)$ instability harder.

Let v's output after the reduction be some b_1. By the selection of the size of Wait, the number of votes received at v for $\not{b}_1$ ($\neq b_1$) must be some $X_0 \leq Cf$.

Now, consider the next reduction at time τ_1, that flips the value of the output to $\not{b}_1$. The number of votes for $\not{b}_1$ may have increased by some X_1 nodes who were not counted among the X_0 above. (Some, or all of the old X_0 may have changed their vote too.) Note, that these X_1 nodes voted b_1 right after τ_0 (and not $\perp$, nor $\not{b}_1$). By Item 3 of Theorem 2, the votes of these X_1 nodes at τ_1 are authentic. Hence, their votes will not change, by the same theorem. If $X_1 > Cf$ then the next flipping reduction (to output b_1) is delayed Cf time, and there are no more changes in v's output, as shown above. Hence, $X_1 < Cf$.

Let us now consider the next flipping reduction (to output b_1) at τ_2. As in the previous argument, there may now be some new Y_2 nodes who did not support b_1 in the previous flipping reduction, but do support b_1 now, and $Y_2 < Cf$.

So far, we established that the number of supporters of b_1, as well as the number of supporters of $\not{b}_1$ are smaller than Cf. From the code, it is easy to see that the new value of Scanned is selected such that there are no $\perp$ voters in $\mathtt{Ball}_v(2^{\mathsf{Scanned}})$. Hence, at that point, Scanned $\leq 2Cf$. The number of additional reductions possible before the next increase is $O(\log f)$. The lemma then follows from the proof for the increases in the value of Scanned (which follows immediately from the fact that `Cancel` is performed when the value of Scanned changes, and from Theorem 2; details are deferred to the full paper). The following lemma that bounds the "damage" of the Wait mechanism.

Lemma 2. *The Stable Adaptive Majority Algorithm of Fig. 1 is time adaptive.*

The proof of Lemma 2 bears similarities to those of [12,13]. The differences result from the following three mechanisms used in the current algorithm: (a) the algorithms in the previous paper outputs the majority of all the arriving votes, while here the algorithm outputs just those in a certain ball; (b) the current algorithm outputs the majority (in a ball) only when there are non- $\perp$ votes from all the nodes (in the ball); (c) the Wait here may cause v to wait before changing the output to that of the majority. Correspondingly, the current proof needs to show the following: (a) the radius of the ball indeed reaches a size that is larger than $2f$ fast enough (so the majority of the votes in the radius are of correct nodes); (b) the radius of the ball is not too large (since otherwise, authentic votes from all the nodes it in may not be received fast enough; (c) the Wait mechanism does not delay the final output longer than $O(f)$. The proof is deferred to the full paper. Intuitively, within $O(f)$ time all the non-authentic votes disappear by Theorem 2 and authentic votes arrive from all the nodes in the ball of radius $O(f)$ around v. If Scanned starts "small" after the faults, then it can be increased to more than $O(2f+1)$, since the $\perp$ values disappear from that ball in $O(f)$ time. More than f authentic votes in that ball imply a correct output. If Scanned starts large, when the non- authentic votes disappear (in $O(f)$ time) the majority it receives is correct. If v receives "many" non- $\perp$ votes, then the output is correct. Otherwise, $\mathsf{Scanned}_v$ is reduced. Finally, when non- authentic votes disappear, $|\mathsf{Suspects}| \leq f$ and hence, Wait $\leq f$.

5.2 Stable Time Adaptive Self Stabilized Persistent Value

(The proof of Theorem 4): We now move to deal with problems where the input may change not just as a result of faults. In particular, solutions for the the Persistent Value Problem have two modules. One, the Input Correction module, maintains (and changes) the storage (*input*) value (see Definition 4). Here, we only replace the second module- the Output Stabilization Module. The latter uses the above storage value as its input. We assume the use of the Input Correction Module introduced in [12]. The following is assumed for that Module (and proven in [12], given a module that stabilizes the output in $O(f)$ time): If $f < n/2$ then the Input Correction Module never changes the value of a non-faulty node. It changes the value of an incorrect node at most twice: the last of these changes is from the incorrect value to the correct one.

The algorithm presented in this section is a modification of the algorithm of Figure 1. Recall, that each decrease in Scanned in that algorithm may cause the output to change. An additional down side to decreasing Scanned is that if $|\mathtt{Ball}_v(2^{\mathsf{Scanned}})|$ becomes smaller than $2f$, then the output may be incorrect (since the majority in the ball may be the faulty nodes) late after the algorithm was supposed to stabilize. This is why the algorithm of Figure 1 would not have been adaptive had it been used for the Persistent Value Problem. (Recall, that here a vote arriving at v may become $\perp$, and then change value, after $\Omega(n)$ time; we do not want that to cause a reduction in Scanned).

The new algorithm does not cut the value of Scanned every time an arriving broadcast changes its value. Instead, Scanned is reduced only when a constant fraction of the nodes in the ball change their mind. In addition, the algorithm attempts to output in the new ball the same value it outputted the previous time (if any) it used for that ball. Only if a significant number of votes changed in the new ball (from that last time) the output is not the output used the last time that new ball was used. This reduces the instability. Additional informal explanations, as well of the proofs of the following claim, are deferred to the full paper. The pseudo code appears in Figure 2.

Claim. For any i, the second time (after the faults ended, and after the first increase in Scanned after the faults) that $\mathsf{Scanned} = i + 1$, all the votes received at v are authentic.

Lemma 3. *If each input changes only a constant number of times (starting from some time t) in the Stable Persistent Value Algorithm, then the number of times* $\mathsf{Scanned} \neq i$ *gets assigned the value i is bounded by a constant.*

Proof Sketch: First, we claim that the number of times Scanned can be reduced from some $i + 1$ to i is bounded by a constant. Let s be the node such that the change in its ABC broadcast vote caused v to cut Scanned from $i + 1$ to i. First consider the case that the changed vote had not been authentic. By Claim 5.2, this can happen at most twice per value of Scanned.

Now, assume that Scanned is reduced because the number of authentic votes for $\mathbf{OldOutput}(i + 1)$ is below $\frac{1}{4}|\mathtt{Ball}_v(2^{i+1})|$ (or below $\frac{1}{4}|\mathtt{Ball}_v(\mathbf{RRadius})|$).

The first sub-case is when $\mathsf{Scanned} < \log \mathbf{RRadius}_v$. The first time Scanned is reduced from $i + 1$ after the faults, the value of $\mathbf{OldOutput}(i + 1)$ could be one that was set by the adversary (remember the setting of self stabilization). However, in later times this value is one that was set by the algorithm, since we are computing the instability at a period when there are no additional faults. Hence, and by Claim 5.2, every time (starting from the second time) $\mathsf{Scanned} =$ is increased to $i + 1$, the real majority of the `inputs` in $\mathtt{Ball}_v(2^{i+1})$ is the value assigned to $\mathbf{OldOutput}(i + 1)$. This means that at the kth time Scanned is reduced from $i+1$ ($k \geq 3$), at least a quarter of the nodes in $\mathtt{Ball}_v(2^{i+1})$ changed their `input` since the $(k - 1)th$ time. However, by the properties of the input correction module, a node can change its `input` at most twice after the faults (if it is a faulty node, otherwise, it cannot change its value at all). Hence, the number of times Scanned can be reduced from $i+1$ to i is a constant. The second sub-case is when Scanned is reduced from $\log \mathbf{RRadius}_v$ to $\log \mathbf{RRadius}_v - 1$. (The proof is similar to the proof of the previous sub-case.)

Now, consider the case that Scanned is set to i by an increase (and not by a reduction). To be increased again to i, the value of Scanned must first be reduced to $i-1$, since a reduction is performed to a consecutive value. Thus, the proof follows from the proof for reductions. ∎

Finally, notice that v may change its `output` only when it changes the value of Scanned. Since the number of different values for Scanned is $\log \mathbf{RRadius}$, this leads only to an instability of $O(\log \mathbf{RRadius})$ which is $O(\log n)$. By a somewhat more precise analysis we obtain the following improved result. In most of the cases, the proof of the following lemma resembles that of Lemma 3. The main difference is in the case that a decrease in Scanned is due to unauthentic votes that disappear. This can happen only at a faulty node v. A sketch of the proof of the next lemma, highlighting the differences between this proof and that of Lemma 3 is deferred to the full paper, together with the proof of Lemma 5 (which bears similarities to the proof of Lemma 2). Theorem 4 follows from the two lemmas bellow and from the assumptions on the Input Correction Module.

Lemma 4. *The instability of the algorithm of Figure 2 is $O(\log f)$ for a non-faulty node, and $O(\min\{\log n, f\})$ for a faulty nodes.*

```
Broadcast the input value and receive every arriving broadcast of other nodes
                                                   (*possibly, undefined (⊥)*).

Do while Scanned > 0 and
    ( (Scanned < $\log \mathbf{RRadius}_v$
                and $NumVotes(\mathbf{OldOutput}(\text{Scanned}) < \frac{1}{4}|\mathtt{Ball}_v(2^{\text{Scanned}})|)$
      or
      (Scanned = $\log \mathbf{RRadius}_v$
                and $NumVotes(\mathbf{OldOutput}(\text{Scanned}) < \lfloor\frac{1}{2}|\mathtt{Ball}_v(2^{\text{Scanned}})|\rfloor + 1)$
      or
      $\mathbf{OldOutput}(\text{Scanned}) = \perp$ )
    $\mathbf{OldOutput}(\text{Scanned}) \leftarrow \perp$;
    Scanned ← Scanned − 1;
    Cancel the broadcast of every node outside $\mathtt{Ball}_v(2^{\text{Scanned}})$.
If Scanned = 0 then $\mathbf{OldOutput}(\text{Scanned}) \leftarrow$ output ← input;
else output ← $\mathbf{OldOutput}(\text{Scanned})$.

Let $i >$ Scanned be the smallest for which $\exists b \neq \perp | NumVotes(b) > \frac{1}{2}|\mathtt{Ball}_v(2^i)|$ ;
(* If $i$ is not undefined then *) Scanned ← $i$;
Set output to the majority value in $\mathtt{Ball}_v(\text{Scanned})$;
$\mathbf{OldOutput}(\text{Scanned}) \leftarrow$ output;
Cancel the broadcast of every node outside $\mathtt{Ball}_v(2^{\text{Scanned}})$.
```

Fig. 2. Stable Persistent Value Algorithm: actions at node v. NumVotes(b) is the number of votes v is currently receiving by ABC from nodes in $\mathtt{Ball}_v(2^{\text{Scanned}})$ for the value b.

Lemma 5. *The Stable Persistent Value Algorithm Algorithm of Figure 2 is time adaptive.*

5.3 Repeated Faults, Majority Consensus with Persistence, and Repeated Majority Consensus

Previous algorithms for the Persistent Value problem assumed (for the sake of obtaining time adaptivity) that all the faults occurred in one batch, and another batch may occur only after full state stabilization. A useful property of the algorithm of Figure 2 is that this assumption is not necessary. Indeed, we did not use it in the proofs. Given that, it is not difficult to change that algorithm to solve the problem of Majority Consensus with Persistence, and using that solution to solve also the Repeated majority Consensus. We omit the changes required to solve these problems from the extended abstract.

6 Conclusion and Future Work

As claimed above, if the algorithms are allowed to be less adaptive, it is easy to change them to have a lower instability, to match the more generalized lower bound of Section 3.

In the extended abstract, we assumed that the topology of the network is known to every node in advance. To lift this assumption, a node needs to detect that some broadcasts it receives are claimed to be arriving from nodes that do not actually exist. We deffer this in the full paper.

We studied the instability for the case that the freedom was in the time till stabilization. It may be interesting to combine that with the freedom to decide what is correct, as studied in [1]. This may be especially interesting since it was demonstrated in [2] that multiple possible input values complicate their problem, while this does not seem the case here.

We studied instability in the context of Consensus (and in the context of Persistence). Instability is expensive in other contexts as well. For example, when a network changes, the routing changes. Instability in the routing tables causes routed messages to loop. It is hoped that the understanding gained here will prove useful for increasing the stability for other problems.

A common criticisms against self stabilizing algorithms is that they do not provide much guarantees on the output of nodes until the stabilization. The current paper provides some such guarantees. It would be interesting to find which additional such guarantees are possible.

References

1. Dolev, S., Rajsbaum, S.: Stability of long-lived consensus. In: JCSS (2003)
2. Davidovitch, L., Dolev, S., Rajsbaum, S.: Consensus continue? Stability of multi-valued continuous consensus! In: GETCO 2004, Amsterdam, pp. 21–24 (October 4, 2004)

3. Dolev, S., Gouda, M., Schneider, M.: Memory requirements for silent stabilization. In: PODC 1996, pp. 27–34 (1996)
4. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: STOC 1993, San Diego, CA, pp. 652–661 (1993)
5. Ghosh, S., Gupta, A., Herman, T., Pemamraju, S.V.: Fault-containing self-stabilizing algorithms. In: PODC 1996 (1996)
6. Ghosh, S., He, X.: Fault-containing self-stabilization using priority scheduling. IPL 73(3-4), 145–151 (2000)
7. Ghosh, S., Pemmaraju, S.V.: Tradeoffs in fault-containing self-stabilization. WSS, 157–169 (1997)
8. Bui, A., Datta, A.K., Petit, F., Villain, V.: State-optimal snap-stabilizing PIF in tree networks. WSS, 78–85 (1999)
9. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
10. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and Pseudo-Stabilization. Distributed Computing 7(1), 35–42 (1993)
11. Dayan, J., Kobett, J.J., Kutten, M.: Control of Surge Drums by Different Types of D. D. C. Algorithms. IJT 10(4) (1972)
12. Kutten, S., Patt-Shamir, B.: Stabilizing Time Adaptive Protocols. TCS 220(1), 93–111 (1999)
13. Kutten, S., Patt-Shamir, B.: Adaptive Stabilization of Reactive Tasks. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328. Springer, Heidelberg (2004)
14. Afek, Y., Kutten, S., Yung, M.: The Local Detection Paradigm and its Applications to Self Stabilization. TCS 186(1–2), 199–230 (1997)
15. Azar, Y., Kutten, S., Patt-Shamir, B.: Distributed Error Confinement. In: PODC 2003, Boston (July 2003)
16. Kutten, S., Peleg, D.: Fault-local distributed mending. J. of Alg. 30, 144–165 (1999)
17. Kutten, S., Peleg, D.: Tight Fault Locality. SIAM J. on Comp. 30(1), 247–268 (2000)
18. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. CACM 17(11), 643–644 (1974)
19. Afek, Y., Bremler, A.: Self-stabilizing unidirectional network algorithms by power-supply. In: SODA (1997)
20. Bremler-Barr, A., Afek, Y., Schwarz, S.: Improved BGP Convergence via Ghost Flushing. In: INFOCOM 2003 (2003)
21. Burman, J., Herman, T., Kutten, S., Patt-Shamir, B.: Time-Adaptive Majority Consensus. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974. Springer, Heidelberg (2006)
22. Breitling, M.: Modeling faults of distributed, reactive systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 58–69. Springer, Heidelberg (2000)
23. Fischer, M.j., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
24. Arora, A., Zhang, H.: LSRP: Local Stabilization in Shortest Path Routing. In: DSN 2003, pp. 139–148 (2003)
25. Zhang, H., Arora, A.: Guaranteed fault containment and local stabilization in routing. Computer Networks 50(18)
26. Afek, Y., Dolev, S.: Local Stabilizer. J. Par. & Dist. Comput. 62(5), 745–765 (2002)
27. Dolev, S., Herman, T.: SuperStabilizing Protocols for Dynamic Distributed Systems. C.J.TCS 1997 4 (1997)

28. Parlati, G., Yung, M.: Non-exploratory self-stabilization for constant-space symmetry-breaking. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 26–28. Springer, Heidelberg (1994)

A Appendix

Definitions for Broadcast with Error Confinement

Definition 6. *A protocol P is said to be an* error-confined *protocol for task Π if for any execution with behavior β (possibly, containing a fault) there exists a legal behavior β' of Π 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 β_v and a suffix $\overline{\beta'_v}$ of β'_v such that $\overline{\beta_v} = \overline{\beta'_v}$.

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.

The broadcast task is defined as follows.

Broadcast (BCAST

Input actions: $\texttt{inp}_s(b)$, done at node $s \in V$, for b in some set D. Node s is called the *source*.

Output actions: $\texttt{outp}(b)$, required at every node $v \in V$, where $b \in D \cup \{\perp\}$.

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