Journal of Parallel and Distributed Computing **62**, 945–960 (2002)
doi:10.1006/jpdc.2001.1824

Scalable Self-Stabilization

Sukumar Ghosh[1] and Xin He

Department of Computer Science, University of Iowa, Iowa City, Iowa 52242
E-mail: ghosh@cs.uiowa.edu, xhe@cs.uiowa.edu

Received February 1, 2000; accepted January 11, 2002

This paper presents a methodology for a synchronous non-reactive distributed system on a tree topology to stabilize from a k-faulty configuration in a time independent of the size n of the system. In the proposed methodology, processes first measure and compare the sizes of the faulty *regions*, and then use this information to schedule actions in such a way that the size of the faulty regions progressively shrink, until they completely disappear. We demonstrate that when k processes fail, the stabilization time is $O(k^2)$. Apart from its applicability to a wide class of problems, the proposed method achieves scalability with a low space complexity of $O(\Delta.(\Delta.k + \log_2 n))$ per process, where Δ is the maximum degree of a node.

1. INTRODUCTION

A *self-stabilizing* distributed system guarantees spontaneous recovery from an arbitrary bad configuration that may be reached by a failure or a perturbation in the system. We assume that faults or perturbations instantaneously corrupt the state of one or more processes, but do not affect the programs in any way. In most of the stabilization algorithms for distributed systems, the time required to stabilize has no connection with the severity of the failure, as a result, the time to recover from a single failure may be as large as the time to recover from a massive failure. This is undesirable. The goal of a scalable self-stabilizing system is to ensure that the recovery time indeed depends only on the severity of the failure, and is independent of the size of the network.

Define a k-faulty configuration as one that is reached by the failure of k processes from a legitimate configuration. It is quite possible that a particular system configuration is reachable by k faults from one legitimate configuration, and l faults ($k \neq l$) from another legitimate configuration. In such cases, the number of failures will be designated by the minimum of (k, l). If the worst-case *stabilization time* from

[1] This research was supported in part by the National Science Foundation under Grant CCR-9901391. An earlier version of this paper was presented at the 4th Workshop on Self-Stabilizing Systems held jointly with ICDCS '99.

0743-7315/02 $35.00

an arbitrary k-faulty configuration is independent of the size of the network, then the self-stabilizing system will be called *fault-scalable*, or simply *scalable*. A system is *fault-containing* [6], when it recovers from every single failure in $O(1)$ time. A special version of scalable stabilization, called *time-adaptive stabilization*, was introduced in [9] to design a persistent bit protocol, in which the stabilization time is proportional to the number of faults.

Although the system can stabilize to any one of the legitimate configurations, it is often desirable, and sometimes expected, that recovery leads to one of the nearest legitimate configurations. In particular, when failures are believed to be minor in nature (i.e., $k \ll n$, the total number of processes), and there is no ambiguity about which processes are faulty, it is desirable that non-faulty processes remain unperturbed during recovery. For this purpose, [6] introduced a new metric called *contamination number* defined by the maximum number of processes that can change their states during recovery from a faulty configuration. Ideally, only the faulty processes should restore their states to return to the legitimate configuration. When this is possible, we will say that recovery is ''tight'', and the contamination number equals the number of faulty processes. However, this is not always true. The study of scalable self-stabilization is the study of techniques by which the user can set an upper bound on metrics like stabilization time, space complexity, and contamination number, commensurate with the magnitude of the failure, and independent of n. We limit our study to non-reactive systems only.

The difficulty of constraining the recovery time in stabilizing distributed systems has been described in [6]. Dolev and Herman [5] presented their superstabilization protocol, which is a stabilizing protocol with the additional guarantee of fast convergence when starting from a legitimate configuration the system undergoes a topology change. Refs. [5,6] emphasize single faults and do not address fault-scalability. In a related work, Kutten and Patt-Shamir [9] presented a transformer that converts a synchronous non-reactive persistent-bit protocol into an equivalent stabilizing protocol whose stabilization time is $O(k)$, k being the number of faulty processes. They use state replication and local voting for this purpose, which result in high space ($O(n^2)$ per process) and communication complexity. Another transformer is presented by Afek and Dolev [1] for reactive systems, but once again, the solution involves high space and communication complexity, combined with a global reset method that takes over when local stabilization fails. Compared to these methods, our method achieves fault-scalability with a much lower space complexity, but at the expense of a higher stabilization time. An asynchronous version of Kutten and Patt-Shamir's earlier result is briefly reported in [10].

The proposed methodology works for synchronous models of non-reactive systems on tree topologies. Our approach leads to a worst-case stabilization time $T(k) = O(k^2)$ rounds from a k-faulty configuration, although for certain classes of problems, the stabilization time will be simply $O(k)$. The additional space complexity is $O(\Delta.(\Delta.k + \log_2 n))$ per process (where Δ is the maximum degree of a node) for any value of k. The contamination number depends solely on the location of the faulty processes, and in some cases it can be as large as $n/2$. The ability to recover to the original legal configuration is also determined by the location of the faulty regions. The paper is organized as follows: Section 2 introduces the model and the notations

to be used. Section 3 presents the general idea behind our method. Section 4 gives an outline of the implementation on a tree topology. Section 5 contains an analysis of the algorithms, including the various performance issues. Finally, Section 6 contains some concluding remarks. An appendix illustrating the applicability of our methodology for some example problems appears at the end.

2. BACKGROUND

2.1. Model, Notations, and Definitions

Consider a tree of n processes. Let Δ designate the maximum degree of any node representing a process. A process i is called a *leaf* if its degree is 1, and we say that *leaf*(i) is true.

Each process communicates with its neighbors using the locally shared memory model. A global state or configuration of the system is a tuple consisting of the states of all the processes. We use s_i to represent the state of process i. We consider a synchronous model of computation, in which the execution of the protocol is organized in rounds. In each round, every process obtains the values of the variables of its immediate neighbors, and executes the protocol simultaneously as a single atomic step. Note that the information exchange only occurs at the beginning of a round. The modification to a local variable will not be visible to its neighbors until next round.

We define a *computation* to be a sequence of global states $(S_0, S_1, ...)$. The global states or configurations are indexed, starting from S_0. The simultaneous execution of the protocol by all the enabled processes in a round changes the state S_i to state S_{i+1}. We define $S_i \prec S_j$ if both S_i and S_j occur in the same computation and S_i occurs earlier than S_j.

Define the *distance* between two configurations S_i and S_j as the number of processes whose states need to be modified to reach one from the other. A fault or a perturbation changes the state of a system from *legitimate* to *illegitimate*. A measure of the extent of this failure is proposed below.

DEFINITION 2.1. A configuration will be called *k-faulty*, if its *smallest distance* from some legitimate configuration is k.

Note that this concept of fault is insensitive to the history of the system. It is more relative than absolute. Thus, if a massive failure leads the system to a configuration that is at a unit distance from another legitimate configuration, then that configuration will be viewed as 1-faulty.

2.2. White and Black Processes

Since our system is non-reactive, in a legitimate configuration, no process is eligible to execute an action. We call such processes *white*. A process with an eligible action will be called *black*, and the execution of the corresponding action will change its color to white. A legitimate configuration has only white processes, but in an arbitrary configuration, some processes may be black.

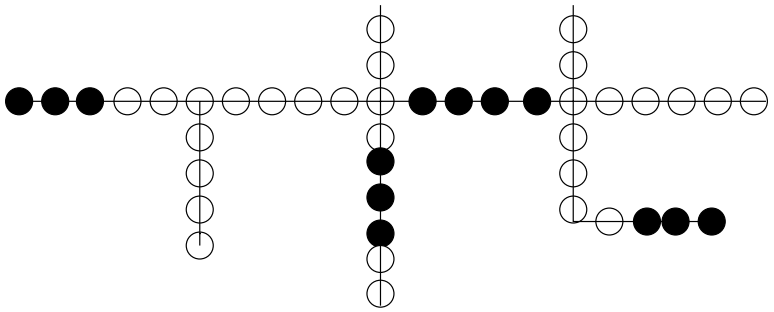


FIG. 1. An illegitimate configuration with black and white regions.

A single failure at a process will change its color from white to black. Depending on the problem and the legitimacy predicate, one or more of the neighbors of the faulty process may also turn black. Thus, a failure at any process i can cause *at most* $(1 + \Delta)$ white processes (which includes i and its Δ neighbors) to turn black. These black processes define a *black region* of size 1. In general, a black region consists of a maximal set of contiguous black processes. Such a region has the property that, by allowing a subset of these processes to execute their actions, their colors become white without affecting the colors of the neighboring processes. The number of black processes that will execute such actions determines the size of the black region, which, for this particular example, is one.

Similarly, *in general*, a white region consists of a maximal set of contiguous white processes. In an illegitimate configuration, there will be black processes bordering the white and black regions (Fig. 1). Due to actions taken by these black processes, the size of the regions will increase or decrease. When the size of a region decreases due to the actions of a black border process, we will include that black process into the region. In a legitimate configuration, there will be only a single white region with no bordering black process. The identification of the boundaries of white or black regions will depend on the problem and the protocol. We refer the readers to the three examples in Appendix A.

3. THE MAIN IDEA

3.1. Shrinking Regions

Assume that due to the failure of one or more processes, multiple regions have been created in the system. Due to the inadequacy of global knowledge, the non-faulty processes bordering a region may sometimes falsely believe that they are faulty, causing them to execute actions that can increase the extent of failure. Our method uses an intelligent scheduling mechanism, using which, in a time independent of the size of the network, and commensurate with the magnitude of the failure, a legitimate configuration is restored by meeting the following two goals:

Goal 1. The number of black regions is reduced to 0.
Goal 2. The number of white regions is reduced to 1.

Note that these two goals are not equivalent. A configuration may have two or more white regions and no black region (see Example 2 in Appendix A).

Informally, the size of a region is determined by the number of processes in that region. A more precise definition is as follows:

DEFINITION 3.1. The *size* of a region is the number of processes that have to change their states so that the region disappears, and the processes in that region merges with those in a neighboring region. The boundaries of a region are defined by the leaf and the border processes.

DEFINITION 3.2. For any region, a *border process* is a black process, whose actions reduce the number of processes belonging to the region.

It is important to define or identify actions that shrink the sizes of regions by undoing the effect of failures. An existing algorithm that is known to be stabilizing, may not have such an action for every type of failure. In such cases, appropriate recovery actions need to be defined and added to the existing actions of the processes. From this perspective, the proposed methodology can be viewed as a transformer that converts a stabilizing solution into a scalable solution.

Given a distributed system and its legitimacy predicate, an illegitimate configuration will consist of a single black region, or several black and white regions. In case there are multiple regions, a pair of adjacent regions can be (i) black and white, or (ii) white and white. To restore legitimacy, we will use the following strategies:

Strategy 1. Each border process of a black region will schedule recovery actions to *reduce* the size of the black region.

Strategy 2. Border processes at the boundary between two or more white regions will schedule recovery actions *on the basis of the relative sizes of the regions* around them, so that the size of the white region of the *smallest size*[2] will be reduced. This will be called the *shortest region first* strategy. We will say that the shortest region "yields" to a neighboring region of larger size, or a neighboring region of larger size "takes over" the shortest region.

Repeated application of the above two strategies will lead to the following results:

LEMMA 3.1. *All black regions belonging to the initial configuration eventually disappear.*

Proof. Every action by a border process of a black region reduces the size of the black region, and no action by any other process increases its size. ∎

Lemma 3.1 implies that in a bounded number of actions, the system configuration will reduce to a set of white regions only.

[2] When two adjacent white regions have equal sizes, process id's of the neighboring border processes will be used as tiebreakers.

THEOREM 3.1. *Starting from a configuration consisting of multiple white regions only, the* shortest region first *strategy eventually reduces the number of white regions to one.*

Proof. Starting from a configuration consisting of white regions only (which is eventually reached per Lemma 3.1), construct a maximal chain of contiguous white regions $(0, 1, 2, \ldots, i-1, i)$ in the order of decreasing sizes. The last region i must either be a boundary region, or be surrounded by regions of larger sizes. In either case, the length of this chain must be finite. Note that there may be multiple such maximal chains in a given configuration of the system.

The *shortest region first* strategy guarantees that actions in each round reduce the size of region i, until this region disappears, i.e., merges with a neighboring white region. Repeated application of this strategy will help reduce the number of white regions to one, and the system will reach a legitimate configuration. ∎

4. IMPLEMENTATION

4.1. Main Structure

A process can determine its color by examining the states of its immediate neighbors. This can be completed in $O(1)$ rounds.

The identification of the border processes will depend on the problem specification, and the legitimacy predicate. To implement Strategy 1, once a black process learns that it borders a black region, it immediately schedules an action that shrinks the size of the black region. Multiple border processes may schedule their shrinking moves at the same time, however, care must be taken so that only non-neighboring border process execute such moves in the same round.

To implement Strategy 2, border processes of neighboring white regions will be required to compare their sizes, before the process bordering the shortest white region executes a shrinking move to reduce its size. This will be done using a *signaling mechanism* explained in the next subsection. The life of a typical process is outlined in Fig. 2. The above program is executed by each process in every round.

```
repeat
update the color;
if color = black then determine if it is a border process fi;
if it is a border process
        then execute moves per strategies 1 and 2
        else
        respond to probes in the system per the signaling mechanism
fi;
forever
```

FIG. 2. The main program for every process.

4.2. The Signaling Mechanism

Here, we outline a method for comparing the sizes of adjacent white regions. The probing method resembles *echo depth sounding*, and works as follows. Once two adjacent border processes i and j of adjacent white regions identify each other, they send out *waves* to probe the sizes of their respective regions using depth-first search (DFS). The DFS wave is carried forward by white processes inside the region, and is continued until the other border processes of that region are found. Note that the task of locating the other boundaries of that region reduces to locating the nearest black processes or leaf processes in that region. When this is done, an *echo* of the wave is carried back to the initiator. This echo contains the *count* of the processes visited by the wave using DFS.

While this simple mechanism will work, it has one major weakness: the time required to compare the size of two regions will depend on the size of the bigger region. This can be as large as $(n-1)$ for a single fault, and will prevent the stabilization time to be independent of the size n of the network, that has been our original goal. To overcome this shortcoming, we will use an *incremental probing* method. In this method, instead of each border process independently sending out a wave to explore the nearest boundary of its region, adjacent border processes will coordinate with one another to send out probes of *predefined* probing distances. The probing distance acts as a *lease*, and with each *new process* visited using DFS, the probing distance is decremented, until the distance reduces to zero, and the lease expires. Initially, all the adjacent border processes start with a probing distance $x = 2$. For any region, regardless of whether exploration of all the boundaries using DFS is complete, the echo of the probe will come back after the expiry of the lease. If all the processes in the region are visited by the probe, then the echo returns the *size* of that region, otherwise a failure is reported to the initiator by returning a $\perp$.

Now, if every probe sent out in the different directions fails to complete the DFS explorations of the respective regions, then the *probing distance is doubled*. The technique is similar to *binary search*. The comparison of the region sizes using probes terminates, when the DFS exploration of *at least one region* is completed. The following outcomes are possible:

- Exactly one probe returns a size, but all other probes return $\perp$. In this case, the region whose probe returned the size is the smallest.
- More probes than one return the values of the sizes of their respective regions. In this case, these regions are smaller than the remaining regions whose probes returned $\perp$. Once the sizes are known, the smallest one can be identified by a direct comparison of these sizes. If two sizes are equal then the identifiers of the border processes will be used as a tiebreaker.

An outline of the size comparison procedure is presented in Fig. 3. The procedure is simultaneously executed by each set of adjacent border processes of white regions. Here, *ready* is a boolean flag that is set to true whenever the border process is ready to send out the probe. At the end of each probing session, the value of *ready* is reset to false.

```
begin
    x := 2 (set the probing distance to 2)
    repeat
    while for all adjacent border processes ready ≠ false
        do ready := false
    {at this point ready = false for all the adjacent border processes}
    ready := true
    initiate DFS probe with the given distance x;
    wait for all the echoes to come back;
    if all the echoes return false (⊥) for every process
        then (the probing distance is too small)
        x := 2x
    fi;
    ready := false;
    until at least one echo returns a size

    determine which region has the smallest size;
    if my region has the smallest size
        then execute a (shrinking) action
    fi;
end
```

FIG. 3. The signaling mechanism for comparing the sizes of white regions.

There are several algorithms for self-stabilizing DFS token circulation in the published literature. We can, for example, use the protocol proposed by Petit and Villain [11] on a tree topology. For DFS, the time required to traverse any subtree containing k nodes is $2k$ rounds. Additionally, Petit's algorithm uses $(\Delta + 2)$ states per process, where Δ is the maximum degree of a node. The only modification that we need is to load the token with a integer, that will return the size of the region to the initiator when the traversal is complete, or a -1 (representing $\perp$) when the traversal has to be prematurely terminated following the expiry of the lease.

Concurrent Probing. Now, take a second look at the maximal chain of white regions $1, 2, 3, \ldots, i$ in the order of decreasing sizes. In a tree topology, each of these regions can have two or more borders. Since no one has global knowledge, it is conceivable that a region j belonging to this chain may expand due to the local action of one border process, but contract due to the local action of another border process. As a result, one may wonder about the real progress of the computation, as perceived by the growth and shrinkage of the regions. Additionally, the overlapped actions initiated at multiple borders can cause the measure of the size of a region to become stale, before a shrinking action is executed. Fortunately, regardless of such phenomenon, the *last region i* in the chain can only shrink at each of its borders that are not leaf processes. Thus, the basic principle of Strategy 2 remains in force, guaranteeing the progress of computation, and convergence towards a legitimate configuration.

5. ANALYSIS OF THE ALGORITHM

5.1. Time Complexity

Let L be the legitimacy predicate for the original problem, which is defined in terms of the output variables. The new variables introduced for the implementation of the algorithm constitute the *state variables*. The color of a process will depend only on the output variables, and not on the state variables. The predicate L' describes the overall legitimate state, that includes both the output and the state variables.

If L is true, we say the system has reached *output stabilization*. If L' is true, then the *state variables* have also stabilized. Once the output variables have stabilized leading to the formation of one white region, the DFS probing is unnecessary, so the state stabilization is quiescent. The DFS probe is triggered by black processes bordering adjacent white regions.

THEOREM 5.1. *$L \rightsquigarrow L'$ in $O(n)$ rounds.*

THEOREM 5.2. *A black region of size k disappears in $O(k)$ rounds.*

Proof. Regardless of the problem specification, a border process of a black region can identify itself in $O(1)$ rounds. Since in each round thereafter, the size of the black region reduces by 1, it takes at most $O(k)$ rounds for the black region to completely disappear. ∎

Note. In fact, due to multiple non-neighboring processes simultaneously executing their moves, the time for a black region to disappear will be $O(D)$ where D is the diameter of the black region.

THEOREM 5.3. *Consider a maximal chain of white regions $0, 1, 2, \ldots, i-1, i$ in the order of decreasing sizes. Then the smallest region i will shrink and merge with a neighboring white region is $O(S^2(i))$ rounds, where $S(i)$ is the size of the smallest region i.*

Proof. Using probing distances $2, 4, 8, 16, \ldots$, it takes $O(S(i))$ rounds for a number of adjacent border processes (i.e., a border process of i and its neighboring border processes) to identify region i as the region of smallest size. This is followed by a move that will shrink the size of the region by at least one. Therefore, in

$$O(S(i) + (S(i) - 1) + (S(i) - 2) + \cdots + 2 + 1),$$

i.e., $O(S^2(i))$ rounds, the region i will disappear and merge with a neighboring white region. ∎

THEOREM 5.4. *If a failure hits k adjacent processes, and the region formed by the faulty processes is smaller than the remaining white regions, then the stabilization time is $O(k^2)$ rounds.*

Proof. If the faulty region is a black region, then the system will stabilize in $O(k)$ rounds (Theorem 5.2). The stabilization time will be worse, when the region is white.

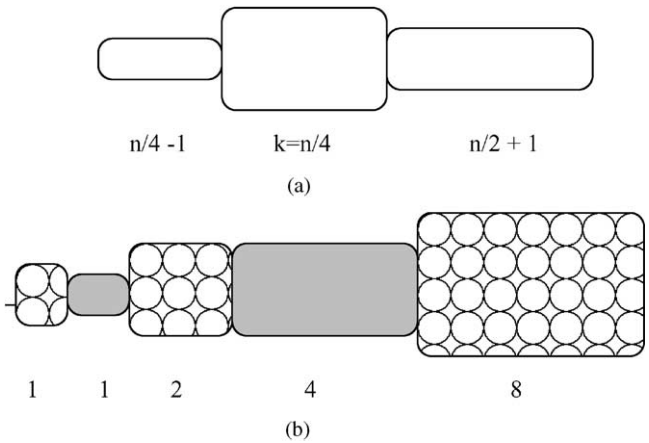


FIG. 4. All regions are white. (a) The white region at the middle takes over the smaller white region before yielding to the largest white region. (b) A bad scenario.

The result immediately follows from Theorem 5.3, since the k-faulty region is the smallest white region. ∎

If the faulty processes form a single region that is larger than one of the remaining white regions, then the stabilization time will be determined by the size of the smallest region that will shrink in the final stage. Consider the faulty configuration in Fig. 4a. Here, the white region at the middle created by the faulty processes will first take over the neighboring white region whose size is smaller than k, before shrinking and restoring stability. The stabilization time in such cases is still $O(k^2)$, although the constant might be bigger. This also leads to the issue of contamination. During stabilization, the processes belonging to the smaller region have to temporarily change their states (to be consistent with the affected region) before stabilizing to the original configuration. Such contaminations are an unavoidable part of the proposed algorithm.

Now, consider that the k faulty processes are not contiguous, i.e., they do not form a single faulty region. How does it affect the stabilization time? Assume that the faulty processes have formed r disjoint white regions $0, 1, 2, \cdots, r-1$, and the number of processes in region i is k_i. Thus,

$$k = k_0 + k_1 + k_2 + \cdots + k_{r-1}.$$

If each white region k_i is bordered by white regions of larger size, then the recovery time for that region is $O(k_i^2)$, and the entire system will stabilize in time $T(k)$ rounds, where

$$\begin{aligned} T(k) &= O(k_0^2 + k_1^2 + k_2^2 + \cdots + k_{r-1}^2) \\ &= O(k^2). \end{aligned}$$

In fact, due to the synchronous operation of the schedulers, the various regions will start shrinking simultaneously, and the stabilization time is $O(k_{max}^2)$, where, k_{max} is the maximum size of the white region formed by the faulty processes.

A bad scenario corresponds to the existence of a chain of white regions of sizes $1, 1, 2, 4, 8, 16, \ldots$, the odd numbered regions being created by failures, and the even numbered ones being the leftover white regions (Fig. 4b). In this case, the stabilization will take place as follows: the first region will take over the second one, the sum total of these two will take over the third one, and so on. Assume that there are $(1 + r)$ odd-numbered white regions created by the failure of k processes. Then,

$$\begin{aligned} k &= 1 + 2 + 8 + 32 + 128 + \cdots + (r+1)\textit{terms} \\ &= 1 + 2/3 \cdot (4^r - 1) : \end{aligned}$$

In this case, the stabilization time $T(k)$ from the k-faulty configuration will be

$$\begin{aligned} T(k) &= O(1^2 + 2^2 + 4^2 + 8^2 + 16^2 + \cdots + (2r+1))\textit{terms} \\ &= O(1 + 4/3 \cdot (4^{2r} - 1)) : \end{aligned}$$

It follows from the above that $T(k) = O(k^2)$. In fact, due to the synchronous operation of the schedulers, the recovery will be faster that what has been shown in the above estimate. We conjecture that from every possible k-faulty configuration, the system will stabilize in $O(k^2)$ rounds.

Per Theorem 5.1, once L is restored, the state stabilization is completed in $O(n)$ rounds.

5.2. *Space Complexity*

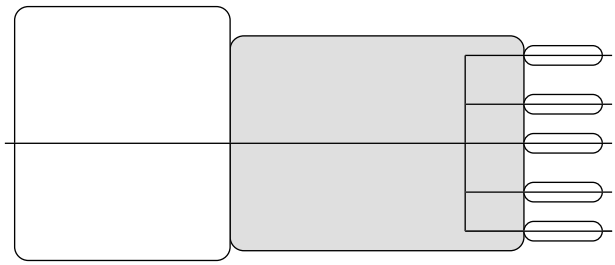
The space complexity is determined by the variables required in the DFS probing. Using [11], the number of variables is $(\Delta + 2)$ per process per probe. Since probes can be concurrently sent out by every border process, and there can be at most $k.\Delta$ border processes for the k-faulty processes in the region, the space complexity will be $O(k.\Delta^2)$. This will take care of all query-response variables that might be needed to complete the probes. Additionally, since process identifiers are used to break ties while comparing the sizes of adjacent regions, an extra space of $O(\Delta \log_2 n)$ per process will be required. Therefore, the overall space complexity is $O(\Delta.(k.\Delta + \log_2 n))$ per process.

Note that the space complexity depends on k, so the space requirement decreases as the Extent of the failure decreases. During stabilization, the faulty regions progressively shrink, until their sizes reduce to zero. This makes our algorithm memory adaptive, when dynamic memory management is available.

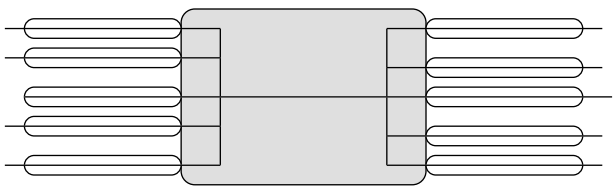
5.3. *Contamination*

The degree of contamination is determined by the number of non-faulty processes that might have to change their output variables before the system reaches a stable configuration, and is called the contamination number. Note that the study of contamination is meaningful, only when the system is destined to return to its *original* configuration. We focus on a few cases here.

Case 1. Assume that the faulty processes form one contiguous black region of size k. Since black regions shrink regardless of their sizes and the characteristics of their neighborhood, no contamination is foreseen.



(a)



(b)

FIG. 5. An illustration of contamination during stabilization. In (a) the system recovers to the original configuration, but in (b), it is not so, even if $k \ll n$.

Case 2. Assume that the faulty processes form a single white region of size k (Fig. 5a). In this case, this region, before shrinking, can take over one or more adjacent regions of size k or smaller, before yielding to a white region of size greater than or equal to $n/2$. In this specific case, the contamination number is $n/2$, but the system is guaranteed to recover to its original configuration. However, regardless of the value of k, the ability to recover to the *original* legal configuration will depend on the location of the faulty processes. For example, in Fig. 5b, even if k is a small fraction of n, the system will not recover to the original legal configuration, since the size of each of the adjacent regions is smaller than k.

Case 3. When the faulty processes form a single white region of size k, and $k > n/2$, the system does not recover to the original configuration, so the estimation of contamination is meaningless.

6. CONCLUSION

The classification of processes into black and white provides a general framework for addressing the problem of stabilization where the recovery time scales with the number of failures, and not with the size of the network. This method has been illustrated on tree networks only. The case of general networks requires additional investigation. Some cases of cyclic topology cannot be directly handled using this

approach. For example, in a cyclic topology, if failures turn every process black, then there will be no border process that can be entrusted with initiating the recovery.

Given a stabilization protocol, sometimes new actions have to be added to implement scalable self-stabilization as explained in Examples 2 and 3 of Appendix A. Readers are cautioned that unless our methodology is used, these new actions can potentially prevent the system from reaching stability when used along with the actions of the original protocol.

The recovery mechanism proposed in the paper depends on the map of the black and white regions, which is related to the number of faulty processes. We demonstrate that, the stabilization time scales with the number of faulty processes, and is independent of the size of the network. Moreover, the stabilization time from a configuration containing two or more regions of k contiguous faults separated by regions of size $k+1$ or more, will be the same as that from a single faulty region of size k.

There are many systems in which it is impossible to have adjacent white regions in an illegitimate configuration. In such cases, stabilization only requires application of Strategy 1, which can be done in $O(k)$ time. Only when there is a possibility of adjacent white regions being present in an illegitimate configuration, the worst case figures of stabilization and contamination are applicable.

Another form of local stabilization addressed in these contexts is k-stabilization [3]. This ensures that if less than k failures occur (for a known k), then the stabilization time is $O(k^2)$. Without a prior estimate of k, this method can be expensive in terms of time, since no performance benefit is guaranteed when the actual number of failures is $<k$.

Apart from general applicability to a wide class of problems, our method is space-efficient. Furthermore, since only a small fraction of the black processes initiate the recovery by comparing the sizes of their local regions, the communication complexity, measured by the number of shared registers read or written in each round, is low. The communication overhead monotonically decreases with the number of regions. Since the state stabilization is quiescent, the communication complexity reduces to zero when a legitimate configuration is reached. This is in contrast with the method proposed in [9], where an active broadcast is used, and the communication complexity remains unchanged regardless of whether the system reaches a legitimate configuration.

Our method does not cost any extra memory in the legitimate state, which is of significance when dynamic memory allocation is used. An asynchronous version of our algorithm for a more restricted class of topology (chain of processes) appears in [7].

APPENDIX A

We present here three example systems, illustrate the formation of black and white regions, and discuss about the applicability of the methodology presented in this paper.

EXAMPLE 1. This example is about maximal matching on a chain of processes. Each process i should find a neighboring matching process j and set its pointer

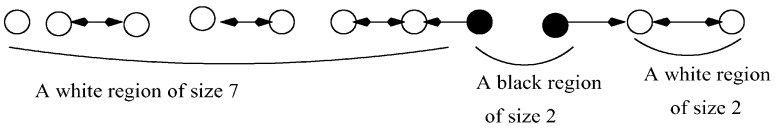


FIG. 6. An illegitimate configuration for the system of Example 1.

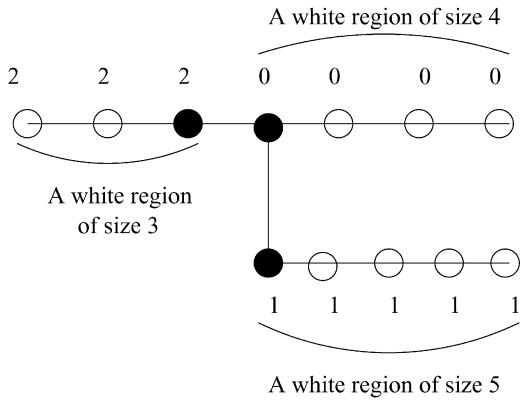


FIG. 7. An illegitimate configuration for the system of Example 2.

towards that process. A process is allowed to remain single, only if there is no unmatched neighbor — in this case, the process will reset its pointer to *null*. A stabilizing algorithm for maximal matching appears in [8].

Figure 6 shows an illegitimate configuration for this system. It has two white regions separated by a black region. The configuration is 2-faulty. As soon as the black processes execute moves, the system recovers to a legitimate configuration.

Note that, in this system, there will not be adjacent white regions. The system returns to a legitimate configuration as soon as the black regions disappear. This expedites the recovery, since stabilization is possible in $O(k)$ time, k being the number of faulty processes.

EXAMPLE 2. Consider the tree network of Fig. 7. Each process has a variable x whose value belongs to the set $\{0, 1, 2\}$. In a legitimate configuration, we want the values of each of these processes to be identical. For stabilization only, the following protocol (for every process i) would work:

$$\textbf{do } \exists j \in N(i) : x(j) = x(i) \oplus_3 1 \rightarrow x(i) := x(j) \textbf{od}.$$

However, to apply our methodology, the above protocol is not adequate. To realize this, consider a 1-faulty configuration in which the faulty process is in state 1, and every other process is in state 0. Using the above protocol, the system can only recover to a legitimate configuration in which the state of every process is 1, and the time to reach this configuration will depend on the size of the network! This is undesirable.

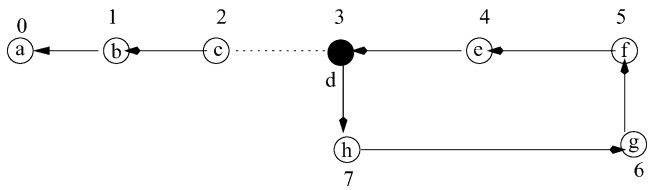


FIG. 8. An illegitimate configuration for the spanning tree of Example 3.

To apply our methodology, we modify the protocol (for process i) as follows:

$$\textbf{do} \exists j \in N(i) : x(i) \neq x(j) \rightarrow x(i) := x(j) \textbf{od}.$$

Now, there are *three* white regions adjacent to one another. Interestingly, the black processes at the border of these regions do not form a black region, since actions taken by them will not cause the region to disappear. These black processes serve as the border processes of these regions, and for this reason, we include them in the white regions. The black color can be attributed to the mutual inconsistency among the three white regions. The occurrence of adjacent white regions is typical in consensus-type protocols.

EXAMPLE 3. The final example is that of a stabilizing spanning tree construction, based on the well-known protocol by Chen *et al.* [4]. Each process has two variables: a parent variable P points to a parent of that node, and a level variable L that is set to $L(P) + 1$.

The example in Fig. 8 shows that, due to one process d accidentally corrupting its pointer P, the system has entered an illegitimate configuration since the parent pointers formed a directed cycle. The original stabilization protocol in [4] is as follows:

$$\begin{array}{ll}
(L(i) \neq n) \wedge (L(i) \neq L(P(i)) + 1) \wedge (L(P(i)) \neq n) & \rightarrow L(i) := L(P(i)) + 1, \\
(L(i) \neq n) \wedge (L(P(i)) = n) & \rightarrow L(i) := n, \\
(L(i) = n) \wedge (\exists k \in N(i) : L(k) < n - 1) & \rightarrow L(i) := L(k) + 1; P(i) := k.
\end{array}$$

If the faulty process is able to reset its parent pointer to the correct value, then the system will return to a legitimate configuration in one step. Interestingly, the available actions in the protocol do not leave room for such an action! The recovery is initiated after every process in the directed cycle executes the action $L \neq L(P) + 1 \rightarrow L := L(P) + 1$. This makes the value of L grow up to the size of the network n, following which the errand process resets its parent pointer. Accordingly, the stabilization time depends on n.

To apply the proposed methodology, we need to modify the existing protocol by adding at least one additional action:

$$\begin{array}{l}
(L(i) \neq L(P(i) + 1)) \wedge (L(i) < n - 1) \wedge \exists j \in N(i) : L(j) \\
\quad = min\{L(k) : k \in N(i)\} \rightarrow P(i) := j; L(i) := L(j) + 1.
\end{array}$$

Furthermore, since by definition, the root has a level 0, and we want the disjoint segments to merge with the root segment in a consistent manner, we need to *define any white region containing the root as the region of largest size*, regardless of the

number of processes contained in that region. This will enable every other white region to merge with the white region containing the root, using the shortest-first strategy.

For the current example, the black process *d* (which is a black region with only one process in it) will execute a move per strategy 1, and restore the original configuration.

ACKNOWLEDGMENTS

We acknowledge the constructive criticisms from the reviewers that led to substantial improvements in the content as well as the presentation of the paper. We also thank Sriram V. Pemmaraju for his help with the performance analysis of the algorithms.

REFERENCES

1. Y. Afek and S. Dolev, Local stabilizer, *in* "Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems," pp. 74–84, 1997.
2. J. Beauquier, S. Delaet, S. Dolev, and S. Tixeuil, Transient fault detectors, *in* "DISC'98," pp. 62–74, 1998.
3. J. Beauquier, C. Genolini, and S. Kutten, Optimal reactive *k*-stabilization: The case of mutual exclusion, *in* "Proceedings of the 18th Annual Symposium on Principles of Distributed Computing," pp. 209–218, 1999.
4. N.S. Chen, H.P. Yu, and S.T. Huang, A self-stabilizing algorithm for constructing spanning trees, *Inform. Process. Lett.* **39** (1991), 147–151.
5. S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems, *Chicago J. Theoret. Comput. Sci.* **3**(4) (1997).
6. S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju, Fault-containing self-stabilizing algorithms, *in* "Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing," pp. 45–54, 1996.
7. S. Ghosh and X. He, Scalable self-stabilization, *in* "Proceedings of the Third Workshop on Self-Stabilizing Systems" (published in association with the 19th IEEE International Conference on Distributed Computing Systems ICDCS'99), pp. 18–24, 1999.
8. S.C. Hsu and S.T. Huang, A self-stabilizing algorithm for maximal matching, *Inform. Process. Lett.* **43** (1992), 77–81.
9. S. Kutten and B. Patt-Shamir, Time-adaptive self stabilization, *in* "Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing," pp. 149–158, 1997.
10. S. Kutten and B. Patt-Shamir, Asynchronous time-adaptive self-stabilization, *in* "Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing," pp. 319, 1998.
11. F. Petit and V. Villain, Time and space optimality of distributed depth-first token circulation algorithms, "DIMACS Workshop on Distributed Data and Structures," pp. 91–106, Carleton University Press, Princeton, NJ, 1999.