ELSEVIER

Information Processing Letters 59 (1996) 281-288

Information Processing Letters

An exercise in fault-containment: Self-stabilizing leader election

Sukumar Ghosh *, Arobinda Gupta

Department of Computer Science, The University of Iowa, Iowa City, IA 52242, USA

Received 5 September 1995; revised 29 April 1996
Communicated by F.B. Schneider

Abstract

Self-stabilizing algorithms are designed to guarantee convergence to some desired stable state from arbitrary initial states arising out of an arbitrarily large number of faults. However, in a well-designed system, the simultaneous occurrence of a large number of faults is rare. It is therefore desirable to design algorithms that are not only self-stabilizing, but also have the ability to recover very fast from a bounded number of faults. As an illustration, we present a simple self-stabilizing leader election protocol that recovers in $O(1)$ time from a state with a single transient fault on oriented rings. Only the faulty node and its two neighbors change their state during convergence to a stable state. Thus, the effect of a single fault is tightly contained around the fault. The technique for transforming a self-stabilizing algorithm into its fault-contained version is simple and general, and can be applied to other problems as well that satisfy certain properties.

Keywords: Self-stabilization; Fault-containment; Leader election; Fault-tolerance; Distributed computing

1. Introduction

The design of self-stabilizing distributed algorithms has emerged as an important research area in recent years. The correctness of self-stabilizing algorithms does not depend on initialization of variables, and a self-stabilizing algorithm converges to some predefined stable state starting from an arbitrary initial state. Self-stabilizing algorithms are thus inherently tolerant to transient faults in the system. Many self-stabilizing algorithms can also adapt dynamically to changes in the network topology or system parameters.

The area of self-stabilization was first introduced in a well-known paper by Dijkstra [2]. Since then, self-stabilizing algorithms have been designed for a large number of problems. Schneider [6] presents an excellent survey of the basic principles of self-stabilization, along with several examples of self-stabilizing algorithms.

Our current research is motivated by two simple observations regarding self-stabilizing algorithms. The first observation is that self-stabilizing algorithms typically do not differentiate between an arbitrary global state and a global state that is "almost stable". The implication is that, even though an "almost stable" state can sometimes be converted into a stable state by a few moves by a few selected processes, often the algorithm allows many moves and requires the cooper-

* Corresponding author. Email: ghosh@cs.uiowa.edu. This author's research was supported in part by National Science Foundation under grant CCR-9402050.

PII S0020-0190(96)00121-4

ation of many more processes, before a stable state is reached.

The second observation is that once a stable state has been reached, it is more likely that occasional transient faults occur only in a small number of processes rather than in an arbitrarily large number of processes. Thus, once a stable state has been reached, transient faults are more likely to lead to an "almost stable" state in practice, rather than an arbitrary state.

These two observations are the primary motivations behind the current exercise. Since faults are rare in a well-designed system, it is desirable to build systems that satisfy the following two criteria:

- (*Stability*) The system must be self-stabilizing, and
- (*Efficiency*) During recovery from a single transient fault[1], only a small number of processes will be allowed to execute recovery actions. All other processes in the system will remain unaffected, and execute normal actions.

There are two consequences of the efficiency aspect. The first relates to fault-containment, which is equivalent to "building a firewall" around the faulty process during the recovery phase. This can also be viewed as an effort to control contamination of the non-faulty processes by the faulty one, and thus, an effort to partially mask the effect of the fault from any higher level protocol. The second is the issue of recovery time. It is expected that the recovery from a single fault would take place following the shortest path in the state space, and that the time required to recover from a single failure will be substantially less than the worst-case stabilization time following an arbitrary failure.

In this paper, we take a step in that direction by presenting a self-stabilizing algorithm for leader election on an oriented ring. The algorithm converges to a state with a single leader in a finite number of steps starting from any arbitrary state. In addition, if there is only a single transient fault in the system, the algorithm converges to a stable state with only the faulty process and its two neighbors making a small constant number of moves to change their states. Thus, the algorithm converges from a state with a single transient fault in only $O(1)$ time. As an illustration, we modify an earlier self-stabilizing algorithm for leader election on a unidirectional ring by Lin and Ghosh [5] to derive the fault-contained self-stabilizing algorithm. The technique used for this modification is simple, and can be used to automatically derive fault-contained self-stabilizing algorithms for many other problems on oriented rings.

In a related work, Dolev and Herman [3] address the issue of designing self-stabilizing algorithms that converge very fast from a small number of topology changes in the network. However, their method relies on an interrupt scheme to signal a topology change to the processes directly affected by the change, and hence cannot be extended directly to yield rapid convergence from transient faults where no such fault-detection mechanism to send interrupts is available. Arora and Kulkarni [1] have presented algorithms in which certain critical transitions are executed only in predefined *safe* states, even in the presence of faults. However, their algorithms tolerate only fail-stop failures and recoveries, and hence, are not self-stabilizing. In a different setting, Gopal and Toueg [4] have considered the problem of contamination of correct processes by a faulty process in broadcast and multicast communication protocols.

The rest of the paper is organized as follows. Section 2 describes a self-stabilizing algorithm for leader election on a unidirectional ring designed earlier by Lin and Ghosh. Section 3 presents our algorithm for leader election on an oriented ring, which is based on the algorithm of Lin and Ghosh. Section 4 presents a proof of correctness of the algorithm. Finally, Section 5 contains some concluding remarks.

2. Leader election on oriented rings

Lin and Ghosh's algorithm selects a leader among N processes connected in a unidirectional ring. Let the processes be numbered from 0 to $N-1$ such that there is an edge in the ring from process $(i-1) \bmod N$ to process $i \bmod N$ for $0 \leqslant i < N$. The process $(i-1) \bmod N$ is said to be the *predecessor* of process $i \bmod N$ in the ring. For simplicity, in the rest of this paper, we will drop the explicit reference to modulo N, and a reference to a process j for any j will actually mean a reference to the process $j \bmod N$. The processes are assumed to have unique ids. The id for process i is denoted by id_i. The process with the max-

[1] A single transient fault can arbitrarily corrupt the state of one or more variables belonging to a single process.

Program for process i:
do
$[S_1]$ $(id_i > max_i) \vee (id_i = max_i \wedge dist_i \neq 0)$
$\vee (id_j \neq max_i \wedge dist_i = 0)$
$\rightarrow max_i := id_i; dist_i := 0;$
$[S_2]$ [] $(dist_{i-1} + 1 < N) \wedge (id_i < max_{i-1})$
$\wedge \neg (max_i = max_{i-1} \wedge dist_i = dis_{i-1} + 1)$
$\rightarrow max_i := max_{i-1};$
$dist_i := dis_{i-1} + 1;$
$[S_3]$ [] $((dist_{i-1} + 1 \geqslant N)$
$\vee (id_i > id_{i-1} \wedge id_i \geqslant max_{i-1}))$
$\wedge \neg (max_i = id_i \wedge dist_i = 0)$
$\rightarrow max_i := id_i; dist_i := 0;$
od

Fig. 1. Lin and Ghosh's leader election algorithm.

imum id is selected as the leader. Each process i has two variables, max_i and $dist_i$. The algorithm of Lin and Ghosh is shown in Fig. 1.

Let the maximum id of any process in the ring be K. Let a *stable state* of the system be defined as follows.

Definition 1. A *stable state* of the system is a state in which the following conditions hold:

(i) For all processes i, $max_i = K$, and

(ii) if j is the process with id K, then $dist_j = 0$. For any other process $i \neq j$, $dist_i = 1 + dist_{(i-1) \bmod N}$.

It has been shown by Lin and Ghosh that starting from an arbitrary state, the algorithm reaches a stable state in a finite number of steps. The process i with $max_i = id_i$ and $dist_i = 0$ declares itself the leader. We next show how to modify this algorithm to obtain a fault-contained self-stabilizing leader election algorithm that converges in O(1) time from a single transient fault. In the rest of this paper, we will refer to the guard of statement S_j of process i as $G_j(i)$ and the action of statement S_j of process i as $A_j(i)$.

3. Fault-contained leader election

Consider an oriented ring that allows bidirectional communication between processes across each link. On such a ring, we present a self-stabilizing leader election algorithm that converges in O(1) time from a single transient fault in the system. Recall that a single transient fault can arbitrarily corrupt the state of a single process. We assume that the ring remains oriented in the presence of faults. Thus, the orientation of the ring is fixed, and is not achieved by any underlying self-stabilizing ring orientation protocol. For simplicity, we also assume that the guarded statements are executed atomically.

We first investigate the behavior of Lin and Ghosh's algorithm under a single transient fault. It is easy to show that the following conditions hold after a single transient fault in the system.

Condition 2. Starting from a stable state, if a single transient fault occurs at a process i, then at least one of $G_1(i)$, $G_2(i)$, or $G_3(i)$ is true after the fault.

Condition 3. Starting from a stable state, if a single transient fault occurs at a process i, then a stable state can be reached again with the executions of the guarded statements in process i only.

Note that since the guards at a process i depend only on the local variables of i and $(i-1)$, a fault at a process i can enable a guard at process i and $(i+1)$, but not at any other process. Since the above condition implies that a fault at process i always enables a guard at i, a first attempt at a simple algorithm that exhibits the desired behavior under a single transient fault can be outlined as follows:

Program for process i:
do
Process i has an enabled guard
if
no guard is enabled in process $(i-1) \bmod N$
$\rightarrow$ *execute the corresponding action;*
[] *some guard is enabled in process* $(i-1) \bmod N$
$\rightarrow$ *skip;*
fi
od

If process i has an enabled guard, i checks if process $(i-1)$ has any enabled guard. If not, and there is indeed a single transient fault, the fault must be at i. Hence process i executes the corresponding action until no guard is enabled at i. If process $(i-1)$ has one or more enabled guards, then process i waits for all guards to be disabled at process $(i-1)$. If there is indeed just a single fault in the system, the fault must be at $(i-1)$ and hence, the execution of the corresponding actions by $(i-1)$ will bring the system back to a stable state.

One problem in implementing the above algorithm is to provide a mechanism for a process i to check the status of the guards of its predecessor. We present a simple two-way communication scheme for this purpose. Each process i has two variables, q_i and a_i. The variable q_i is used to *question* process $(i-1)$ about the status of its guards. The variable a_i is used to *answer* to a similar question by process $(i+1)$. The variable q_i can take the values $\{0, 1\}$ and a_i can take the values $\{\perp, 0, 1\}$. We interpret $q_i = 1$ as i asking a question to $(i-1)$, and $q_i = 0$ as i not asking any question. In response to a question by $(i+1)$, $a_i = \perp$ implies that i has not answered the question. We interpret $a_i = 0$ to mean that i has no enabled guard, and $a_i = 1$ to mean that i has an enabled guard. Thus, the value taken by a_i can be expressed by the function f_i as shown below. For notational convenience, we define $G_p(i) = G_1(i) \vee G_2(i) \vee G_3(i)$.

$$f_i = \begin{cases} \perp & \text{if } q_{i+1} = 0, \\ 1 & \text{if } q_{i+1} = 1 \text{ and } G_p(i) \text{ is true}, \\ 0 & \text{if } q_{i+1} = 1 \text{ and } G_p(i) \text{ is false}. \end{cases}$$

In a stable state, $q_i = 0$ and $a_i = \perp$ for all i. The algorithm is shown in Fig. 2. By $A_p(i)$ in guarded statement S_5, we mean an appropriate action (chosen from $A_1(i)$, $A_2(i)$, or $A_3(i)$) corresponding to the enabled guard. A brief explanation of the guarded statements follows.

If $G_p(i)$ is true, the global state may be corrupted due to a single transient fault at i. Guarded statement S_4 enquires about the status of $G_p(i-1)$ by setting q_i to 1. Statement S_5 considers the case when $G_p(i)$ is true but $G_p(i-1)$ is not. Hence, the global state cannot be due to a single fault at $i-1$, and hence i executes an action corresponding to $G_1(i)$, $G_2(i)$, or $G_3(i)$ depending on which of the guards are true. Statement S_6 sets a_i correctly if it is not already so. Guarded statement S_7 sets q_i to 0 if $G_p(i)$ is false, as the global state surely cannot be one arising from a single fault at i and hence i has no need to take any action at all.

Though the algorithm achieves the stated claims under a single transient fault correctly, it is however no longer self-stabilizing in the presence of arbitrary failures. In particular, if the system starts in a state such that all processes have an enabled guard, the system will deadlock, as every process will wait for its predecessor to move. For example, consider an initial state in which for every process i, $dist_i = 0$, $q_i = 1$, $a_i = 1$, and $max_i = X$, where X is a number greater than the maximum id of any process in the ring. It can be easily verified that in such a state, no guard of the algorithm in Fig. 2 is enabled for any process. Hence, no process makes any move, even though the global state is clearly not stable. However, a simple solution to the problem is proposed below.

Program for process i:
do
$[S_4]$ $G_p(i) \wedge (q_i = 0 \wedge a_{i-1} = \perp) \rightarrow q_i := 1;$
$[S_5]$ [] $G_p(i) \wedge (q_i = 1 \wedge a_{i-1} = 0) \rightarrow$ execute $A_p(i)$;
$[S_6]$ [] $(a_i \neq f_i) \rightarrow a_i := f_i;$
$[S_7]$ [] $\neg G_p(i) \wedge (q_i \neq 0) \rightarrow q_i := 0;$
od

Fig. 2. A first attempt at a fault-contained version.

Note that a single fault can cause guards to be enabled in at most two consecutive processes. Therefore, the presence of a sequence of three processes, $(i-2)$, $(i-1)$, and i with enabled guards implies that the system cannot be in a state with a single fault, and hence process i can move irrespective of the status of the guards in process $(i-1)$. Thus, we introduce for each process i, an additional 1-bit variable c_i that simply copies the value of a_{i-1} if $a_{i-1} = 1$, else takes the value 0. The value taken by c_i can be expressed by the function g_i shown below.

$$g_i = \begin{cases} 1 & \text{if } a_{i-1} = 1, \\ 0 & \text{if } a_{i-1} \neq 1. \end{cases}$$

The modified algorithm is shown in Fig. 3. We will refer to this algorithm as the protocol P_f in the rest of this paper. A brief explanation of the guarded statements follow.

Guarded statement S_4 is the same as that in Fig. 2. Statement S_5 is similar to that in Fig. 2, excepting that if $G_p(i)$ is true, i not only executes an action corresponding to one of the enabled guard $G_1(i)$, $G_2(i)$, or $G_3(i)$ if $G_p(i-1)$ is false, but also if $G_p(i-1)$ is true but c_{i-1} is set to 1, indicating that $G_p(i-2)$ is true. Statement S_6 sets both c_i and a_i correctly if not already so. Finally, statement S_7 is the same as statement S_7 in Fig. 2.

Program for process i:
do
$[S_4]$ $G_p(i) \wedge (q_i = 0 \wedge a_{i-1} = \perp)$ $\rightarrow$ $q_i := 1$;
$[S_5]$ [] $G_p(i) \wedge (q_i = 1)$
$\wedge((a_{i-1} = 0) \vee (a_{i-1} = 1 \wedge c_{i-1} = 1))$ $\rightarrow$ execute $A_p(i)$;
$[S_6]$ [] $(a_i \neq f_i) \vee (c_i \neq g_i)$ $\rightarrow$ $c_i := g_i$;
$a_i := f_i$;
$[S_7]$ [] $\neg G_p(i) \wedge (q_i \neq 0)$ $\rightarrow$ $q_i := 0$;
od

Fig. 3. The final version of the algorithm, P_f.

4. Proof of correctness

The proof of correctness of the protocol P_f is divided into two parts, a proof of self-stabilization, and a proof of fault-containment from a state with a single transient fault. Note that a single transient fault at a process i can arbitrarily corrupt the state of one or more of the variables max_i, $dist_i$, q_i, a_i, or c_i.

4.1. Proof of self-stabilization

We first define a stable state of the protocol P_f.

Definition 4. A stable state of P_f is a state in which the following conditions are satisfied.
(i) $G_p(i)$ is false for all i, and
(ii) $q_i = 0$, $a_i = \perp$, and $c_i = 0$ for all i.

It is easy to see that a stable state of P_f implies a stable state of Lin and Ghosh's algorithm. Also, no guard of P_f is enabled in a stable state.

We first prove the partial correctness of P_f. To prove this, we show that if no guard is enabled in P_f, the system is in a stable state. We first prove the following lemmas.

Lemma 5. *If $G_p(i)$ remains true for any process i, then $q_i = 1$ within a finite number of steps.*

Proof. Assume that there exists a process i such that $G_p(i)$ remains *true*, and $q_i = 0$. Depending on the value of a_{i-1}, two cases are possible.

If $a_{i-1} = \perp$, guard $G_4(i)$ is true. Also, since $G_p(i)$ remains true, and since a_{i-1} can change from $\perp$ to a value not equal to $\perp$ only if $q_i = 1$, $G_4(i)$ remains true until q_i is set to 1 by an execution of $A_4(i)$. Hence, we only need to show that the action $A_4(i)$ is executed within a finite number of steps, that is, no process can execute an infinite number of moves before an execution of $A_4(i)$. Define a *synchronization move* as a move that changes the values of the variables q, a, or c of a process. Thus, the moves in S_4, S_6, and S_7 are synchronization moves. Since a synchronization move cannot enable any of the guards G_1, G_2, or G_3 for any process, the termination of Lin and Ghosh's algorithm implies that the number of executions of the statement S_5 by any process is finite. It is easy to verify that if no process executes S_5, the number of executions of synchronization moves by any process is also finite. Hence, $A_4(i)$ is executed within a finite number of steps and q_i is set to 1.

If $a_{i-1} \neq \perp$, then guard $G_6(i-1)$ is true. Also, since $q_i = 0$ and q_i can change to 1 only if $a_{i-1} = \perp$, $G_6(i-1)$ remains true until a_{i-1} is set to $\perp$ by an execution of $A_6(i-1)$. The same arguments as above can be used to show that $A_6(i-1)$ is executed within a finite number of steps, setting a_{i-1} to $\perp$. Then, as in the first case, q_i is set to 1 within a finite number of steps. □

Lemma 6. *If no guard of P_f is enabled in the system, then for every process i, $G_p(i)$ is false.*

Proof. Since no guard of P_f is enabled in the system, no further state change can occur in the system. Consider any process i. Let the guards $G_4(i)$ and $G_5(i)$ be written as $G_4(i) \equiv (G_p(i) \wedge X(i))$ and $G_5(i) \equiv (G_p(i) \wedge Y(i))$. That is, $X(i) \equiv (q_i = 0 \wedge a_{i-1} = \perp)$, and $Y(i) \equiv (q_i = 1) \wedge ((a_{i-1} = 0) \vee (a_{i-1} = 1 \wedge c_{i-1} = 1))$.
We will show that if no guard of P_f is enabled in the system, then either $X(i)$ or $Y(i)$ is true. Since $G_4(i)$ and $G_5(i)$ are both false, this will imply that $G_p(i)$ must be false.

Assume that $X(i)$ is false. Then $(q_i = 1 \vee a_{i-1} \neq \perp)$ must be true. Since $G_6(i-1)$ is false, $a_{i-1} = f_{i-1}$.

Therefore, $q_i = 1$ implies that $a_{i-1} \neq \perp$ and $a_{i-1} \neq \perp$ implies that $q_i = 1$. Hence $q_i = 1$ and $a_{i-1} \neq \perp$. If $a_{i-1} = 0$, $Y(i)$ is true. If $a_{i-1} = 1$ and $c_{i-1} = 1$, again $Y(i)$ is true. The only other possibility is the case when $a_{i-1} = 1$ and $c_{i-1} = 0$. We now show that $G_5(i-1)$ is false implies that this case is not possible.

Since $G_5(i-1)$ is false, at least one of the three conjuncts in $G_5(i-1)$ must be false. If $G_p(i-1)$ is false, then $a_{i-1} \neq 1$, as $G_6(i-1)$ is false implies that $a_{i-1} = f_{i-1}$. If $G_p(i-1)$ is true, then by Lemma 5, $q_{i-1} = 1$. Hence, $a_{i-2} \neq 0$, as $G_5(i-1)$ is false. Therefore, $a_{i-2} = \perp$ or $a_{i-2} = 1$. But since $G_6(i-2)$ is false, $a_{i-2} = f_{i-2}$. Therefore, $a_{i-2} \neq \perp$, since $q_{i-1} = 1$. Hence, $a_{i-2} = 1$. But then, $G_6(i-1)$ is false implies that $c_{i-1} = g_{i-1}$, and therefore $c_{i-1} = 1$. Hence, $a_{i-1} = 1$ and $c_{i-1} = 0$ is not possible. Hence, if $X(i)$ is false, $Y(i)$ is true.

Hence, either $X(i)$ or $Y(i)$ must be true. Since $G_4(i)$ and $G_5(i)$ are both false, $G_p(i)$ must be false. □

Lemma 7. *If no guard of P_f is enabled in the system, then for every process i, $q_i = 0$, $a_i = \perp$, and $c_i = 0$.*

Proof. Since no guard of P_f is enabled in the system, by Lemma 6, $G_p(i)$ is false for all i. Since $G_7(i)$ is false, and $G_p(i)$ is false for all i, $q_i = 0$ for all i. Since $G_6(i)$ is false, $a_i = f_i$ for all i. Therefore, $a_i = \perp$ for all i, as we have already shown that $q_i = 0$ for all i (and hence, $q_{i+1} = 0$ for all i). Also, since $G_6(i)$ is false, $c_i = g_i$ for all i. Hence $c_i = 0$ for all i, as $a_i = \perp$ for all i (and hence $a_{i-1} = \perp$ for all i). □

Theorem 8. *The algorithm P_f is partially correct.*

Proof. The theorem follows directly from Lemma 6, Lemma 7, and Definition 4. □

Next we prove the termination of the algorithm. Recall that a *synchronization move* is a move that changes the values of the variables q, a, or c of a process. Thus, the moves in S_4, S_6, and S_7 are synchronization moves. Our algorithm contains two types of moves, synchronization moves, and executions of A_1, A_2, or A_3.

Theorem 9. *Starting from an arbitrary state, the algorithm P_f terminates within $O(N + t_N)$ steps, where t_N is the time complexity of Lin and Ghosh's algorithm.*

Proof. An execution of a synchronization move cannot enable any of the guards G_1, G_2, or G_3 for any process. Hence the number of executions of A_1, A_2, or A_3 in P_f is the same as that in Lin and Ghosh's algorithm.

Starting from an arbitrary state, if no process executes A_1, A_2, or A_3, a process i can execute only a constant number of synchronization moves. Thus the total number of synchronization moves before some process executes A_1, A_2, or A_3 is $O(N)$.

Next we find the maximum number of new synchronization moves that can be caused by an execution of A_1, A_2, or A_3 by a process i. Such an execution of $A_1(i)$, $A_2(i)$, or $A_3(i)$ can cause at most two synchronization moves in i (for setting q_i appropriately if necessary, and for setting a_i correctly if $q_{i+1} = 1$), a single synchronization move in process $(i-1)$ (for setting a_{i-1} correctly if q_i changes), and a single synchronization move in process $(i+1)$ (to correct c_{i+1} if a_i changes). It is easy to verify that no new synchronization move can be caused at any other process because of an execution of $A_1(i)$, $A_2(i)$, or $A_3(i)$. Thus, the total number of new synchronization moves caused by the executions of A_1, A_2, or A_3 is $O(t_N)$.

Hence the algorithm terminates in $O(N + t_N)$ steps. □

4.2. Proof of fault-containment

In this section, we show that starting in a state with a single transient fault, only the faulty process and its neighbors make a constant number of moves before a stable state is reached again. Consider a single transient fault at a process i that corrupts the state of i arbitrarily. We first prove the following lemma. The lemma shows that during convergence from a single transient fault, if $q_j = 1$ and $a_{j-1} \neq \perp$ for any process j, then a_{j-1} and c_{j-1} must be set correctly. We will then use the lemma to prove that starting from a state with a single transient fault at process i, only process i can execute an action of the type A_1, A_2, or A_3 before the system reaches a stable state.

Lemma 10. *Starting from a state with a single fault at a process i, $q_j = 1$ and $a_{j-1} \neq \perp$ for any process j implies that*

(i) $a_{j-1} = f_{j-1}$, *and*
(ii) $c_{j-1} = g_{j-1}$,
that is, a_{j-1} *and* c_{j-1} *are set correctly.*

Proof. In a stable state, $q_j = 0$, $a_j = \perp$, and $c_j = 0$ for all j. Since a fault at i can corrupt only the variables at i, $q_j = 0$, $a_j = \perp$, and $c_j = 0$ for all $j \neq i$ after the fault.

We first consider process i. Since a_{i-2} and c_{i-1} are not affected by the fault, $a_{i-2} = \perp$ and $c_{i-1} = 0$ after the fault. Therefore, $c_{i-1} = g_{i-1}$ after the fault. $a_{i-1} = \perp$ before the fault and its value is unchanged by the fault. Either $q_i = 1$ after the fault, or if $G_p(i)$ is true after the fault, q_i may be set to 1 by an execution of $A_4(i)$. In either case, if $q_i = 1$, $G_6(i-1)$ is enabled and a_{i-1} is set to f_{i-1} from $\perp$ by an execution of $A_6(i-1)$. Note that c_{i-1} is already equal to g_{i-1}. Hence, when $q_i = 1$ and $a_{i-1} \neq \perp$, $a_{i-1} = f_{i-1}$ and $c_{i-1} = g_{i-1}$.

We next consider any process $j \neq i$. If $G_p(j)$ is true at some time during convergence, q_j may be set to 1. However, $q_j = 0$ before the fault and its value is unchanged by the fault at i. Since q_j is set to 1 only if $a_{j-1} = \perp$, therefore, if $a_{j-1} \neq \perp$ subsequently, then process $(j-1)$ must have executed $A_6(j-1)$ after q_j has been set to 1, and therefore both a_{j-1} and c_{j-1} are set correctly. Note that since c_{j-1} is set before a_{j-1}, c_{j-1} is already correct when $a_{j-1} \neq \perp$. □

We next show that only the faulty process and its neighbors can possibly make moves before the system reaches a stable state. To show this, we consider executions of A_1, A_2, A_3, and executions of synchronization moves separately. We first show that only process i can execute an action of the type A_1, A_2, or A_3. We next show that only process i, $(i-1)$, and $(i+1)$ can execute synchronization moves,

Lemma 11. *Starting from a state with a single fault at a process* i, *only process* i *can execute an action of the type* A_1, A_2, *or* A_3 *before the system reaches a stable state.*

Proof. If the fault at i does not change the value of the variables max_i and $dist_i$, $G_p(j)$ is false for all j after the fault. Hence no process will execute A_1, A_2, or A_3, and only synchronization moves will be executed by some processes before the system reaches a stable state. If the fault at process i changes the value of the variables max_i and $dist_i$, only $G_p(i)$ and $G_p(i+1)$ can be true just after the fault. Hence, right after the fault, only processes i and $(i+1)$ can potentially execute an action of the type A_1, A_2, or A_3, depending on which of the guards are true. It is easy to verify that if process $(i+1)$ does not execute A_1, A_2, or A_3, then when $G_p(i)$ is false (after execution of $A_1(i)$, $A_2(i)$ or $A_3(i)$ by process i), $G_p(i+1)$ is also false, and the system is in a stable state. Moreover, $G_p(j)$ remains false for any process j other than i or $(i+1)$ during the executions of $A_1(i)$, $A_2(i)$ or $A_3(i)$ by i. Therefore, it suffices to show that process $(i+1)$ does not execute an action of the type A_1, A_2, or A_3 if $G_p(i)$ is true after the single fault at i.

To show this, we need to show that if $G_p(i)$ is true, $G_5(i+1)$ is necessarily false. Let $X(i+1) \equiv (q_{i+1} = 1 \wedge a_i \neq \perp)$. It is easy to see that if $X(i+1)$ is false, then $G_5(i+1)$ is false. If $X(i+1)$ is true, then by Lemma 10, a_i and c_i are set correctly. Therefore $a_i = 1$, as $G_p(i)$ is true. Also, since the fault is at i, $G_p(i-1)$ is false and a_{i-1} is not corrupted by the fault. Hence, $a_{i-1} \neq 1$. Therefore, $c_i = 0$. But $a_i = 1$ and $c_i = 0$ implies that $G_5(i+1)$ is false. Hence, if $G_p(i)$ is true after a single fault at i, $G_5(i+1)$ is false. □

The leads to the following lemma regarding the synchronization moves.

Lemma 12. *Starting from a state with a single fault at a process* i, *synchronization moves are executed only by* $(i-1)$, i, *and* $(i+1)$. *Moreover, each process executes only a constant number of such moves.*

Finally, the following theorem holds.

Theorem 13. *Starting from a state with a single fault, only the faulty process and its neighbors make a constant number of moves before the system reaches a stable state.*

The theorem follows directly from Lemmas 11 and 12.

5. Conclusions

In this paper, we have presented for an oriented ring with bidirectional communication capabilities, a simple self-stabilizing leader election algorithm that

converges in constant time from a single transient fault. Moreover, the effects of the fault are tightly contained in a very small neighborhood around the fault. We have used a simple technique to derive the fault-contained self-stabilizing algorithm for leader election from an existing self-stabilizing algorithm for leader election. However, the applicability of the technique is not limited to leader election only. For any unidirectional ring, given any self-stabilizing protocol P that satisfies Conditions 2 and 3, the exact same technique can be used to automatically derive a fault-contained self-stabilizing algorithm P_f for the problem on oriented rings with bidirectional communication capabilities. The protocol P_f converges in constant time from a single transient fault with only the faulty node and its two neighbors changing their states during convergence. The technique also suggests the interesting possibility of designing general techniques to convert self-stabilizing algorithms to corresponding fault-contained self-stabilizing algorithms for other classes of problems as well. We are currently investigating this issue.

The price paid to achieve this fast convergence from a single transient fault is an increase in the convergence time from an arbitrary initial state. Even though we have shown that the convergence time of our algorithm from an arbitrary initial state is the same as that of Lin and Ghosh's algorithm in an asymptotic sense, the constant involved in our algorithm is larger than that of Lin and Ghosh's algorithm. However, this increase is not damaging as the number of faults in a well-designed system is usually small, and a single transient fault is more likely than a large number of transient fault that take the system to an arbitrary state. In fact, it can be easily shown that the claims on fault containment hold even in the presence of multiple faults, as long as the distance between two faulty processes is at least 3. In this case, the faults are sufficiently *dispersed* around the ring so as to not affect each other.

Finally, we would like to note that in this paper, we have considered only the time for stabilization of the *entire local state* of the nodes starting from some initial state. However, in certain cases, it may be more important to partition the local state of each node into a *critical* and a *non-critical* part, define stability for each of the parts independently, and ensure that the critical part of the local states of all nodes stabilize very fast, even at the expense of a larger convergence time for the non-critical part (and hence the entire protocol). For example, in the leader election protocol presented in this paper, the variables max_i and $dist_i$ at each node i can be viewed as the critical part of the local state of i and the variables q_i, a_i, and c_i constitute the non-critical part. If the critical part of the local states of all nodes are in a stable state, then the node with the maximum id is the leader and no node will change its *max* and *dist* variables again. Thus, the system will function correctly to any higher level protocol using the elected leader, even though the non-critical part of the states may not have stabilized yet.

Acknowledgements

We thank the anonymous referees for many helpful suggestions that greatly improved the presentation of this paper. We also acknowledge many helpful discussions with Sriram V. Pemmaraju and Ted Herman.

References

[1] A. Arora and S. S. Kulkarni, Masking fault-tolerance from non-masking fault-tolerance, in: *Proc. IEEE Symp. on Reliable Distributed Systems*, 1995.

[2] E.W. Dijkstra, Self-stabilization in spite of distributed control, *Comm. ACM* **17** (1974) 643–644.

[3] S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems, Tech. Rept. TR-95-02, Dept. of Computer Science, The University of Iowa, Iowa City, 1995.

[4] A. Gopal and S. Toueg, Inconsistency and contamination (preliminary version), in: *Proc. ACM Symp. on Principles of Distributed Computing* (1991) 257–272.

[5] X. Lin and S. Ghosh, Maxima finding in a ring, in: *Proc. 28th Ann. Allerton Conf. on Computers, Communication, and Control* (1991) 662–671.

[6] M. Schneider, Self-stabilization, *ACM Comput. Surveys* **25** (1993) 45–67.