Optimal, Nonmasking Fault-Tolerant Reconfiguration of Trees and Rings

Anish Arora*

Department of Computer Science
The Ohio State University
Columbus, OH 43210

Ashish Singhai

Department of Computer Science
University of Illinois
Urbana-Champaign, IL 61801

December 10, 1993

Abstract

We design two programs that maintain the processes of an arbitrary distributed system in a rooted spanning tree and in a unidirectional ring, respectively, in the presence of fail-stop failures and repairs of both processes and communication channels. Our programs are notable as they (i) are fully distributed, (ii) have optimal time complexity, and (iii) demonstrate two different approaches to designing nonmasking fault-tolerant programs.

Keywords: distributed computing, nonmasking fault-tolerance, process and channel failure and repair, reconfiguration, stabilization.

Categories and Subject Descriptors:
C.2.4 [Computer Communication Systems]: Distributed Systems—*distributed applications*, *network operating systems;*
D.1.3 [Programming Techniques]: Concurrent Programming;
D.4.5 [Operating Systems]: Reliability—*fault-tolerance, verification;*
G.2.2 [Discrete Mathematics]: Graph theory—*rings, graph algorithms.*

[0]Tel: 614-292-1836 , fax: 614-292-2911 , email: anish@cis.ohio-state.edu
Supported in part by NSF grant CCR-9308640 and an OSU Seed Grant.

1 Introduction

Cooperation between the processes of a distributed system is often made feasible by organizing the processes in a convenient logical structure such as a ring, a star, or a tree. Such cooperation becomes complicated, however, if faults occur during the computation of the distributed system. Fault occurrences can perturb the logical structure, resulting in a loss of cooperation and thereby necessitating a reorganization of the processes.

The problem of reorganizing the processes of a distributed system into the desired logical structure can be solved at various levels of generality.

- A first solution is to design an *offline* program, that executes after all faults have occurred. Typically, offline programs rebuild the desired structure from scratch, rather than repair the perturbed structure that resulted from the fault occurrences. Moreover, offline programs are often themselves fault-intolerant.
- A better solution is to design an *online* program, that executes after every fault occurrence to repair the perturbed structure. Typically, online programs assume that faults do not reoccur while they compute.
- A still better solution is to design a *realtime* program, that executes after every fault occurrence to repair the perturbed structure within some realtime bound. Typically, realtime programs assume that the realtime bound of their computation is less that the time interval between fault occurrences.
- An ideal solution is to design a *nonmasking fault-tolerant* program, that executes continually in the presence of faults to maintain the structure. Nonmasking fault-tolerant programs do not assume any predictability of fault occurrence and, hence, tolerate further perturbations to the structure that occur while reorganization is in progress.

In this paper we focus our attention on nonmasking fault-tolerant programs.

One approach to designing a nonmasking fault-tolerant program to maintain a structure, say x, is to first compute the set X of all structures that result from perturbing x by any number of fault occurrences (including 0). Then derive the

program so that starting from any program state where the structure is in X, subsequent computation of the program (i) always yields program states where the structure is in X and (ii) reaches within a bounded number of steps a program state where the structure is x. The correctness of this approach follows from the fact that even if faults occur while the program is executing, the program is always in a state where the structure is in X; hence, subsequent computation of the program is guaranteed by (ii) to reach a state where the structure is x.

An alternative approach is to make no assumption about the structure that results from the execution of faults. Instead, ensure that the program is "stabilizing", in the following sense. Starting from an arbitrary program state, subsequent computation of the program converges within a bounded number of steps to a program state where the structure is x. Correct execution of the distributed system then resumes, and continues until a subsequent fault occurrence perturbs the structure, in which case the cycle of convergence repeats.

In this paper we employ both approaches discussed above to present two fully distributed nonmasking fault-tolerant programs that, respectively, maintain the processes in an arbitrary but connected distributed system in a rooted spanning tree and in a unidirectional ring. Specifically, our programs are nonmasking fault-tolerant of the fail-stop failures and repairs of both processes and communication channels.

Our programs are further distinguished by the fact that their time complexity is optimal: Both reach a fixpoint within $O(N)$ rounds of computation, where N is the number of processes of the distributed system that are "up" (i.e., that are currently not stopped due to a fail-stop failure).

Our program for ring maintenance is optimal in another sense: The ring it establishes spans at most $2(N-1)$ communication channels and the number of channels between successive processes in the ring is at most 3. Both these "length" and "dilation" bounds are optimal since there exist distributed systems that have no ring embedding of length less than $2(N-1)$ or dilation less than 3. (For example, consider respectively processes organized in a line and in a 3 by 3 mesh with any two parallel lines deleted that are a unit distance apart. See Figure 0.)

The rest of this paper is organized as follows. In Section 2, we state our assump-

tions and give a formal definition of programs and their nonmasking fault-tolerance. In Section 3, we use the first design approach discussed above to present a nonmasking fault-tolerant program for maintaining a rooted spanning tree. We build upon this program, in Section 4, to present a stabilizing program for maintaining a unidirectional ring. In Section 5, we discuss related work and make some concluding remarks.

2 Preliminaries

2.1 Assumptions

We make the following assumptions about the distributed system whose processes are to be maintained in some structure. The system consists of M processes named 1, ... , M. At each instant, each process is either "up" or "down". Only up processes can execute actions to maintain the desired structure. Henceforth, we use the two terms "process" and "up process" interchangeably. Executing the actions of a process may involve communication with at most one other process, provided the two are "adjacent"; i.e. connected by an up channel. Channels are bidirectional.

Fail-stop failures can arbitrarily change the set of up processes and channels, as long as the processes remain connected. (We assume connectivity for simplicity alone: If the processes become partitioned, then the processes of each partition will be organized into the desired structure.) Repairs can likewise arbitrarily change the set of processes and channels. Repairing a process or channel may involve initializing its state appropriately.

Each process j maintains a set $Adj.j$ of the processes adjacent to j. The specific details of maintaining $Adj.j$ are outside the scope of this paper. One possibility, though, is that each process j periodically checks every potentially adjacent process k and uses a timeout to establish whether k is in $Adj.j$. A nonmasking fault-tolerant program for maintaining $Adj.j$ appears in [2].

2.2 Programs

Each process j has the form:

process j
  owned var ⟨variable declarations⟩ ;
  shared var ⟨variable declarations⟩ ;
  parameter ⟨parameter declarations⟩ ;
  begin
    ⟨action⟩ [] ... [] ⟨action⟩
  end

Thus, j is defined by a set of *variables*, owned or shared, a set of *parameters*, and a set of *actions*. Each of these is defined in some detail next.

Each variable of j has a predefined nonempty domain, and is read only by the actions of j and processes in $Adj.j$. An owned variable is written only by actions of j, whereas a shared variable is written only by actions of j and processes in $Adj.j$.

Each parameter of j ranges over a finite domain. The function of a parameter is to define a set of actions as one parameterized action. For example, let m be a parameter whose value is 0, 1 or 2; then the parameterized action $act.m$ in the action set of a process abbreviates the following set of three actions.

$$act.(m := 0) \;\; [] \;\; act.(m := 1) \;\; [] \; act.(m := 2)$$

Each action of j has the form

$$\langle \text{guard} \rangle \;\; \longrightarrow \;\; \langle \text{assignment statement} \rangle$$

A guard is a boolean expression over the variables and parameters of j, and the variables of processes in $Adj.j$. An assignment statement writes the variables of j and processes in $Adj.j$. Each action of j can access (i.e., read or write) at most one shared variable. (Observe that due to the definition of owned variables, no action of j can write any owned variable of processes in $Adj.j$.)

A *program* p is a finite set of processes. A *state* of p is defined by a value for each variable in each process of p (chosen from the domain of the variable). An action is *enabled* of p at a state iff the guard of the action holds at that state.

A *state predicate* of p is a boolean expression over the variables of p. A state

predicate S is a *fixpoint* of p if for each state of p where S holds, no action of p is enabled in that state. A state predicate T is *preserved* by an action of p iff executing the action, starting from any state of p where the action is enabled and T holds, yields a state where T holds. A state predicate T is *closed* in p iff T is preserved by each action of p.

A *computation* of p is a fair, maximal sequence of steps; in every step, some action in p that is enabled in the current state is executed. Fairness of the sequence means that each action in p that is continuously enabled along the sequence is eventually executed. Maximality of the sequence means that if the sequence is finite then no action in p is enabled in the final state.

2.3 Nonmasking Fault-Tolerance

Let p be a program that maintains a structure x in a distributed system. We first observe that the set of "fault-free" states of p , i.e. the states where the structure is x, can be characterized by a predicate S that is a fixpoint of p. Such a state predicate identifies the *invariant* of p.

We next observe that the set of states that p can reach in the presence of faults can be characterized by a state predicate T that is closed in p [1]. Such a state predicate identifies the *fault-span* of p. Note that the fault-span includes the fault-free states of the program. Examples showing how to design fault-span predicates appear in [3]. These examples employ the view that all classes of faults can be represented as actions that change the program state [3, 4]. (For instance, a fail-stop failure of a process can be represented as an action that marks an up process as being down, and a repair can be represented as an action marks a down process as being up and reinitializes the process.)

We are now ready to give a formal definition of nonmasking fault-tolerance. Let S be the invariant and T be the fault-span of p, such that $(S \Rightarrow T) \wedge (S \not\equiv T)$. (Hence, S is a fixpoint of p and T is closed in p.) Let F be a set of fault actions. For p to be nonmasking-tolerant of F, the following two requirements should be satisfied :

- Closure: T is preserved by the actions of F.
- Convergence: Starting from any state where T holds, each computation of p reaches a state where S holds.

For a more general definition of nonmasking fault-tolerance, we refer the interested reader to [1, 3].

3 Maintaining a Rooted Spanning Tree

In this section, we employ the first of the two design approaches presented in Section 1 to derive a nonmasking fault-tolerant program for maintaining a rooted spanning tree. Recall that in this approach we first compute the set of all structures that result from perturbing the desired structure, in this case a rooted spanning tree, by any number of occurrences of fail-stop failures and repairs.

Let us represent the rooted spanning tree in terms of a *parent* relation between processes; each process j maintains the index of the current parent process of j in the tree. Now, if any number of processes fail-stop, then the resulting graph of the parent relation is a spanning forest of trees. Likewise, if any number of processes are repaired, then assuming that each process initializes itself to be its own parent upon repair, the resulting graph of the parent relation is a spanning forest of trees.

We therefore propose to let the collection of all spanning forests of trees be the set X of structures that is preserved by the program we derive.

To derive our program, we need to address two problems. First, how to handle multiple rooted trees in the graph of the parent relation. And second, how to handle trees that are not rooted, i.e. trees that have a process whose parent is down.

To solve the first problem, we need a mechanism to merge trees. We propose to merge trees by giving precedence to the tree whose root has the highest index, as follows. Each process j maintains a root value denoting the index of the process that j considers to be the root of the spanning tree containing j. If the root value of j is lower than the root value of an adjacent process k, j merges into the tree containing k: j adopts the root value of k as the root value of j and makes k the

parent of j. The net effect of these merge actions is that (i) the root values of the processes along all tree paths from their root to their leaves is always nonincreasing and (ii) eventually the root value of each process is identically the highest index among the indices of all up processes.

To solve the second problem, we need a mechanism to correct the parent of a process without introducing a cycle in the graph of the parent relation. We propose to correct the parent thus by associating a color with each process, as follows. As long as all ancestors of a process j are up, the color of j is green. When a green colored j detects that its parent is down or is colored red, j colors itself red. Also, when a red colored j detects that it has no remaining children, it disowns its parent: it colors itself green, makes itself its own parent, and adopts its index as its root value. The net effect of these actions is that when a process goes down (i) all descendents of that process color themselves red and then (ii) all descendents of that process respectively disown their parent. As a result, the process with the highest index elects itself as a root and colors itself green, following which the remaining processes can join its tree via merge actions.

Observe that merge actions should be allowed to execute between processes j and k only if the color of both processes is green. For, if j merges with k when k is colored red, j will subsequently have to disown k. And, if j merges with k when j is colored red, the resulting state may have a red colored node whose ancestors are all up.

An informal description of our program now follows. Process j maintains the parent, root and color values in variables $par.j$, $root.j$ and $d.i$, respectively. The process has three actions.

The first action is used to color j red; as described above, this action is enabled when $par.j \notin Adj.j \cup \{j\}$ or $col.(par.j) = red$. The second action is used to disown the parent of j; as described above, this action is enabled when each process $k \in Adj.j$ satisfies $par.k \neq j$. The last action is used to merge j into the tree containing k; as described above, this action is enabled when $root.j < root.k \wedge col.k = green \wedge col.k = green$.

Our program, *RST*, for maintaining a rooted spanning tree is therefore:

process $j \ \ (j : 1..K)$

owned var $root.j, par.j, d.j : 1..K;$

$col.j : \{green, red\};$

$Adj.j$: **Set** of $1..K;$

parameter $k : 1..K;$

begin

$col.j = green \ \wedge$

$(par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red) \longrightarrow \quad col.j := red$

$[]$

$col.j = red \ \wedge$

$(\forall k : k \notin Adj.j \ \vee \ par.k \neq j) \quad \longrightarrow \quad col.j, root.j, par.j := green, j, j$

$[]$

$col.j = green \ \wedge \ k \in Adj.j \ \wedge$

$col.k = green \ \wedge \ root.j < root.k \quad \longrightarrow \quad col.j, root.j, par.j := col.k, root.k, k$

end

Figure 1 : Program RST

In the rest of this section, we prove that RST is nonmasking fault-tolerant of fail-stop failures and repairs. We begin by characterizing the fault-span predicate T and the invariant predicate S.

Let $T \equiv$ graph of the parent relation is a spanning forest of trees $\wedge$

$(\forall j : \ j$ is up $\Rightarrow$

$(col.j = red \quad \Rightarrow \quad (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red)) \ \wedge$

$(par.j = j \quad \Rightarrow \quad root.j = j) \ \wedge$

$(par.j \neq j \quad \Rightarrow \quad root.j > j) \ \wedge$

$(par.j \in Adj.j \ \Rightarrow \quad col.(par.j) = red \ \vee \ root.j \leq root.(par.j)))$

Let $S \equiv \ T \ \wedge$

$(\forall j : \ j$ is up $\Rightarrow$

$(col.j = red \quad \Leftarrow \quad (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red)) \ \wedge$

$col.j = green \quad \wedge \quad (\forall k : k \notin Adj.j \ \vee \ root.j = root.k))$

Theorem 1 *At each state of RST where S holds, the graph of the parent relation is a rooted spanning tree.*

Proof: At each state where S holds, T holds and, therefore, the graph of the parent relation is a spanning forest of trees. We show that this forest consists of at most one tree: At each state where S holds, all processes are colored green, have a parent that is up, and have the same root value. It follows that there can be at most one process j satisfying $root.j = j \ \wedge \ par.j = j$ and, hence, at most one tree. □

Lemma 2 *S is a fixpoint of RST.*
Proof: In Appendix 1.

Lemma 3 *T is closed in RST.*
Proof: In Appendix 1.

Theorem 4 *(Closure) T is closed under fail-stop failures and repairs.*
Proof: In Appendix 1.

Theorem 5 *(Convergence) Starting from any state where T holds, every computation of RST reaches a state where S holds.*

Proof: Our proof is in three stages: First, we show that starting from any state where T holds, every computation of RST reaches a state where U holds, $U \equiv (T \wedge (\forall j : j \text{ is up} \ \Rightarrow \ (col.j = red \ \Leftarrow \ (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red))$. Second, we show that starting from any state where U holds, every computation of RST reaches a state where V holds, $V \equiv (U \wedge (\forall j : j \text{ is up} \ \Rightarrow \ col.j = green))$. Finally, we show that starting from any state where V holds, every computation of RST reaches a state where S holds. The details of these three stages appear in Appendix 1.

We measure the time complexity of convergence to S in terms of rounds. A round is a minimal, nonempty sequence of program steps wherein for each process there exists a step where the process either executes an action or has no actions enabled before or after the step.

Stage 1 requires at most N rounds; stage 2 requires at most N rounds; and stage 3 requires at most $2N - 3$ rounds, where N is the number of up processes. It follows that the time complexity of RST is $O(N)$ rounds, which is optimal. □

4 Maintaining A Unidirectional Ring

In this section, we employ the second of the two design approaches presented in Section 1 to derive a nonmasking fault-tolerant program for maintaining a unidirectional ring. Recall that in this approach, we show that regardless of the starting state of the program, subsequent computation of the program converges within a bounded number of steps to a state in which the desired structure, in this case a unidirectional ring, ring is established.

To motivate our program derivation, we first give a constructive proof of

Lemma 6 *A unidirectional ring embedding of length at most* $2(N-1)$ *and dilation at most 3 can be embedded in an arbitrary but connected graph.*

Proof: Given any connected graph, a rooted spanning tree can be embedded in it. We show by structural induction on the height h of the rooted spanning tree, that a unidirectional ring can be embedded in the rooted spanning tree such that the following two properties are satisfied.

1. In the ring embedding, the successor of the root is a child of the root, and the root is the successor of a grandchild of the root.
2. The length of the ring is at most $2(N-1)$ and the dilation of the ring is at most 3.

Base Case ($h = 0$) : The tree consists of a single node, hence a self-loop on the node is a unidirectional ring embedding that satisfies the properties 1-2. (To verify property 1, we let a leaf node be its own child and, hence, its own grandchild.)

Induction Step ($h > 0$) : From the induction hypothesis, for each child c of the root r of the spanning tree, a ring $ring.c$ can be embedded in the subtree rooted at c such that the properties 1-2 are satisfied. From property 1, there exists a node $fc.c$ that is adjacent to c in $ring.c$ and at a distance of ≤ 1 from c. We construct next a ring embedding in the tree rooted at r from the ring embeddings in the subtrees rooted at the children of r. (See Figure 2.)

Consider the children of r in some fixed total order that starts with $fc.r$. (i) Connect r to $fc.r$. (ii) Invert the successor relation of $ring.c$ for each child c of r.

(iii) Connect $fc.c$ of the last child c in the chosen order to r. (iv) For each child c of r, delete from $ring.c$ the edge from $fc.c$ to c, and connect $fc.c$ to the next sibling of c in the chosen order, provided c is not the last child of r. The resulting structure is a ring that satisfies the properties 1-2. □

In accordance with the existence proof above, we propose to derive a program that consists of two layers: The first layer of our program maintains, in a nonmasking fault-tolerant fashion, a rooted spanning tree. The second layer of our program uses this rooted spanning tree to maintain, in a stabilizing fault-tolerant fashion, the desired unidirectional ring.

The first layer can be implemented using the program RST that we presented in Section 3. Hence, we do not elaborate on the first layer here. We merely assume that the following variables describing the rooted spanning tree are available to the second layer of each process j.

- $par.j$, denoting the parent process of j ($par.j = j$ if j is the root process).
- $fc.j$, denoting respectively the first process in some fixed total ordering of the children process of j ($fc.j = j$ if j is a leaf).
- $ns.j$, denoting respectively the next sibling of j in the fixed total ordering of the children process of $par.j$ ($ns.j = j$ if j is the last child of $par.j$).
- $even.j$, a boolean denoting whether j is at an even distance from the root process. (To see that $even.j$ is easily implemented, observe that the actions in program RST can be augmented to maintain a distance variable. The distance value of j is 0, if j is a root process, and is one more than the distance value of $par.j$, if j is not a root process. Hence, $even.j$ can be deduced from the distance variable of j.)

We now describe the second layer. Let is represent the unidirectional ring in terms of a *successor* relation; the second layer in each process j maintains the index of the current sucessor of j in the ring. Following step (i) of the existence proof above, we choose the successor for the root r of the spanning tree to be $fc.r$. Following step (ii) of the existence proof above, we choose the successor for the non-root processes j of the spanning tree, as follows. Since the succesor relation of the ring embeddings of all subtrees is inductively inverted in step (ii), we conclude

that the final value of the successor of j depends solely on whether the depth of j in the spanning tree is odd or even: if j is at an odd depth, then the successor of j is determined by steps (iii) or (iv); and if j is at an even depth, then the successor of j is determined by the inverse of steps (iii) or (iv).

More specifically, given $even.j$, we can uniquely determine the successor $s.j$ for each process j in one step, as follows:

(a) j is at odd depth and $ns.j = j$: from step (iii), $s.(fc.j) = par.j$.

(b) j is at even depth and $ns.j = j$: from step (iii) and the inversion argument made above, $s.(par.j) = fc.j$.

(c) j is at odd depth and $ns.j \neq j$: from step (iv), $s.(fc.j) = ns.j$.

(d) j is at even depth and $ns.j \neq j$: from step (iv) and the inversion argument made above, $s.(ns.j) = fc.j$.

Our stabilizing fault-tolerant program, UR, for the second layer is therefore:

---

process $j \ \ (j : 1 \ldots K)$

owned var $par.j, fc.j, ns.j : 1..K;$

$even.j : Boolean;$

shared var $s.j : 1..K;$

begin

$$
\begin{array}{llll}
even.j & \wedge\ par.j = j & \wedge\ s.j \neq fc.j & \longrightarrow\ s.j := fc.j \\
[]\ \neg even.j & \wedge\ ns.j = j & \wedge\ s.(fc.j) \neq par.j & \longrightarrow\ s.(fc.j) := par.j \\
[]\ even.j & \wedge\ ns.j = j & \wedge\ s.(par.j) \neq fc.j & \longrightarrow\ s.(par.j) := fc.j \\
[]\ \neg even.j & \wedge\ ns.j \neq j & \wedge\ s.(fc.j) \neq ns.j & \longrightarrow\ s.(fc.j) := ns.j \\
[]\ even.j & \wedge\ ns.j \neq j & \wedge\ s.(ns.j) \neq fc.j & \longrightarrow\ s.(ns.j) := fc.j
\end{array}
$$

end

Figure 3 : Program UR

In the rest of this section, we prove that UR is nonmasking fault-tolerant of fail-stop failures and repairs. We begin by characterizing the invariant predicate S and fault-span predicate T.

Let S be the predicate denoting all states where no action of program UR is enabled, and let T be the predicate *true*. Note that, by definition, S is a fixpoint of UR and T is closed in UR as well as fail-stop failures and repairs.

Lemma 7 *At each state of UR where S holds, for each process j, there is some process k (possibly identical to j) such that s.j=k.*

Proof: In Appendix 2.

Lemma 8 *At each state of UR where S holds, for each up process j, there is some up process k (possibly identical to j) such that s.k=j.*

Proof: In Appendix 2.

Lemmas 7 and 8 imply that at each state where S holds the in-degree and out-degree of each process j in the graph of the successor relation is 1. Hence, the graph of the successor relation is a collection of process disjoint rings. We can now prove

Theorem 9 *At each state of UR where S holds, the graph of the successor relation is a ring.*

Proof: We prove by structural induction over the height h of the rooted spanning tree maintained by RST that all processes of a tree of height h belong to the same ring. Let the fact that two up processes k and l belong to the same ring be denoted by $k =_r l$. Note that $=_r$ is an equivalence relation.

Base Case ($h = 0$) : The tree consists of a single process, hence the hypothesis is trivially satisfied.

Induction Step ($h > 0$) : From the induction hypothesis, if processes k and l belong to the subtree rooted at a child c of r, then $k =_r l$. Since action 1 of proces R is not enabled, $s.r = fc.r$, and hence $r =_r fc.r$. Since $h > 0$, if the last child of the root is l then $l \neq r$. Since action 2 of l is not enabled, $s.(fc.l) = r$, i.e. $r =_r l$.

Finally, if c is not the last child of r, $ns.c \neq c$. Since action 4 of c is not enabled, $s.(fc.c) = ns.c$, which implies $c =_r ns.c$.

By transitivity of $=_r$, we have that $r =_r fc.r =_r ns.(fc.r) =_r \ldots =_r l$ where l is the last child of r. Thus, all processes of the tree rooted at r belong to the same ring. □

Observe that each action of *UR* assigns a successor that is at a distance of at most 3. Further, since the graph of the successor relation is a ring that traverses a rooted spanning tree maintained by the actions of *RST*, the length of that ring is at most $2(N-1)$. We conclude the proof with

Theorem 10 *(Convergence): Starting from any state, every computation of UR reaches a state where S holds.*

Proof: Observe that the guards of all actions in *UR* are mutually exclusive, and that the values assigned to the s variables depend only on the variables of *RST*. Further, recall from Theorem 4, that every computation of *RST* eventually reaches a fixpoint. It therefore follows that once *RST* has reached a fixpoint, within one round of computation of *UR*, the values of s variables will not change. In other words, it takes *UR* at most one round of computation to converge after *RST* converges. Thus, the time complexity of maintaining a unidirectional ring in the presence of fail-stop failures and repairs is also $O(N)$. □

5 Related Work and Conclusions

Distributed programs for ring embeddings have been presented previously [5, 6], but unlike our program they do not tolerate fail-stop failures and repairs.

The distributed program of Hélary and Raynal [5] enumerates ring nodes in the order of a depth first traversal of the tree. While this limits the length of the embedded ring to $2(N-1)$, it allows the dilation to be linear in N. Another drawback of their approach is that the ring is formed on node at a time. The message-size complexity of their program is O(N), as messages contain a list of nodes already visited in the tree traversal. By way of contrast, a message passing version of our program will use only messages of constant size.

Rosenstiehl et al. [6] have presented a network of finite state automata, all of which execute in unison to embed a ring of dilation at most 3 in a tree. Again, the ring is formed one node at a time.

Distributed programs for rooted spanning tree embeddings have been studied extensively. Gallagher et al. [7] have presented an elegant solution, but their solution is fault-intolerant. More recently, stabilizing solutions for maintaining a rooted spanning tree have been proposed [8, 9], but these are time-wise less efficient and program-wise more complex than *RST*.

Observe that program *RST* can also be viewed an optimal, nonmasking fault-tolerant program for leader election.

Finally, we would like to note that the requirement of fairness with respect to continuously enabled actions is not necessary for the correctness of programs *RST* and *UR*, but is used only in simplifying the proofs of correctness. Further, there exist refinements of these programs into read/write atomicity programs that are nonmasking fault-tolerant as well. We conjecture that these read/write atomicity programs can be nicely translated into nonmasking fault-tolerant programs that use message passing.

References

[1] A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing", *IEEE Transactions on Software Engineering* 19(11) (1993), pp. 1015-1027.

[2] A. Arora and D. Poduska, "Real-time guarded commands", manuscript (1993).

[3] A. Arora, "A foundation of fault-tolerant computing", *Ph.D. Dissertation*, The University of Texas at Austin (1992).

[4] F. Cristian, "A rigorous approach to fault-tolerant programming", *IEEE Transactions on Software Engg.* 11(1), (1985).

[5] J.-M. Hélary and M. Raynal, "Virtual ring construction in parallel distributed systems", M. Cosnard (ed.), *Parallel Processing*, Elsevier Science Publishers, 1988, pp. 333-345.

[6] P. Rosenstiehl, J. R. Fiksel, and A. Holliger, "Intelligent graphs: networks of finite automata capable of solving graph problems", R. C. Read (ed.), *Graph Theory and Computing*, Academic Press, 1972, pp. 219-265.

[7] R. G. Gallagher, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees", *ACM Transactions on Programming Lang. and Sys.* 5(1) (1983), pp. 66-77.

[8] A. Arora and M. G. Gouda, "Distributed reset", *IEEE Transactions on Computers* (1993), to appear; extended abstract in: *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science 472*, Springer-Verlag (1990), pp. 316-331.

[9] G. Varghese, "Self-stablization by local checking and correction", *Ph.D. Dissertation*, Massachusetts Institute of Technology (1992).

[10] R. Perlman, "An algorithm for distributed computation of a spanning tree in an extended LAN", *Ninth ACM Data Communications Symposium*, Vol. 20, No. 7 (1985), pp. 44-52.

[11] F. B. Bastani, I.-L. Yen, and I.-R. Chen, "A class of inherently fault-tolerant distributed programs", *IEEE Transactions on Software Engg.* 14(10) (1988), pp. 1431-1442.

Appendix 1

Lemma 2 *S is a fixpoint of RST.*

Proof: We show that each action of each process j is not enabled at any state of RST where S holds:

1. S
   $\Rightarrow$ { from the definition of S }
   $par.j \in Adj.j \cup \{j\} \ \wedge \ col.(par.j) = green$
   $\Rightarrow$ { predicate calculus }
   $\neg(par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red)$

$\Rightarrow$

action 1 is not enabled

2. S

$\Rightarrow$ { from the definition of S }

$col.j = green$

$\Rightarrow$

action 2 is not enabled

3. S

$\Rightarrow$ { from the definition of S }

$k \in Adj.j \;\Rightarrow\; root.j = root.k$

$\Rightarrow$

action 3 is not enabled $\Box$

Lemma 3 *T is closed in RST.*

Proof: We show that T is preserved by each action of RST:

1. The only actions of RST that can violate the first conjunct of T are the second and third actions of each process j. Action 2 of process j, however, is enabled only if j is a leaf of some tree, and upon execution removes j from that tree and adds to the forest of spanning trees a tree consisting of only one process, namely j itself. Also, if there is a path in the graph of the parent relation from process k to process j and both k and j are colored green, then it follows from T that $root.k \leq root.j$ and, hence, action 3 of j is not enabled. Therefore, each action of RST preserves the first conjunct of T.

2. The only actions of RST that can violate $col.j = red \;\Rightarrow\; (par.j \notin Adj.j \cup \{j\} \;\vee\; col.(par.j) = red)$ are the actions of process j and $par.j$. However, actions 1 and 2 of j trivially establish this predicate; and action 3 of j preserves this predicate since it is not enabled if $col.j = red$. Likewise, each action of $par.j$ preserves this predicate since no action of $par.j$ is enabled if $col.j = red$.

3. The only actions of RST that can violate $par.j = j \;\Rightarrow\; root.j = j$ or $par.j \neq j \;\Rightarrow\; root.j > j$ are the actions of process j. However, actions 1 and 2 of process j trivially preserve both of these predicates; and action 3 establishes $par.j \neq j$ and increases $root.j$ and thus preserves both of these predicates.

4. The only actions of RST that can violate $par.j \notin Adj.j \ \vee \ col.(par.j) = red \vee root.j \leq root.(par.j))$ are the actions of processes j and $par.j$. However, action 1 of j is enabled only if $par.j \notin Adj.j \ \vee \ col.(par.j) = red$ and thus preserves this predicate; action 2 of j establishes $par.j = j$ and thus preserves this predicate; and action 3 of j increases $root.j$ to $root.(par.j)$ and thus preserves this predicate. Also, action 1 of $par.j$ preserves this predicate since it establishes $col.(par.j) = red$; action 2 of $par.j$ is not enabled in states where this predicate holds, since $par.j$ has a child process; and action 3 of $par.j$ preserves this predicate since it increases $root.(par.j)$. □

Theorem 4 *(Closure) T is closed under fail-stop failures and repairs.*

Proof: When a process j fail-stops at a state where T holds, it can only further fragment the graph of the parent relation, thereby yielding a state where the graph of the parent relation is still a forest of trees. Hence, the first conjunct of T is preserved when j fail-stops. Since the second conjunct of T is universally quantified over up processes, it is trivially preserved when j fail-stops.

Repairing a process j at a state where T holds yields a state where $root.j = j \ \wedge \ par.j = j \ \wedge \ col.j = red$ holds. In such a state, the graph of the parent relation is still a spanning forest of trees, $root.j = j \ \wedge \ par.j = j$ is established for j, the second conjunct of T is established for j and preserved for any children of j (since $col.j = red$). Hence, T is preserved when j is repaired. □

Theorem 5 *(Convergence) Starting from any state where T holds, every computation of RST reaches a state where S holds.*

Proof (continued): 1. For the first stage, we show by induction on $d, 0 \leq d < N$, that each process j whose depth in its tree is at most d satisfies $col.j = red \ \Leftarrow \ (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red)$ after d rounds. It follows that a state where U holds is reached after N rounds.

Base case ($d = 0$): An up process at depth 0 satisfies $col.j = col.(par.j)$, so the claim is trivially true.

Induction case ($d \geq 0$): By the induction hypothesis, each process j whose depth in its tree is at most d–1 satisfies $col.j = red \ \Leftarrow \ (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red)$

after $d-1$ rounds. If j executes no action during round d then the negation of the first guard necessarily holds throughout the round. If j executes action 1 during the round then $col.j = red$ is established. If j executes action 2 or 3 during the round then $par.j \in Adj.j \cup \{j\} \ \wedge \ col.(par.j) \neq red$ is established. It follows that each process j whose depth in its tree is at most d satisfies $col.j = red \ \Leftarrow \ (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red)$ after d rounds.

2. For the second stage, observe that at each state where U holds, each process can only execute its action 2 or 3 and that both of these actions preserve U. We show by induction on $d, 0 \leq d < N$, that each process j whose depth in its tree is more than $N-d$ satisfies $col.j = green$ after d rounds. It follows that a state where V holds is reached after N rounds.

Base case ($d = 0$): The maximum depth of a process in its tree is at most N, so the claim is trivially true.

Induction case ($d > 0$): By the induction hypothesis, each process j whose depth in its tree is more than $N-d+1$ is colored green after $d-1$ rounds. If j executes no action during round d then, since U holds, $col.j = green$ holds throughout the round. If j executes action 2 during the round then $col.j = green$ is established. If j executes action 3 during the round then $col.j = green$ is preserved. It follows that each process j whose depth in its tree is at most d satisfies $col.j = green$ after d rounds.

3. For the third and final stage, observe that at each state where V holds, each process can only execute its action 3 and that action 3 preserves V. We show by induction on e, $0 \leq e < N$, that for all n, $1 \leq n < N$, each process j that is at a distance e in the graph of the adjacency relation from some process whose x value is maximum, satisfies $(Sum\, z : z > x.j \wedge (\exists k :: x.k = z) : 1) \ < \ N - n$ after $e + n - 1$ rounds. It follows that each process j satisfies $(Sum\, z : z > x.j \ \wedge \ (\exists k :: x.k = z) : 1) \ < \ 1$ after $2N-3$ rounds; in other words, each process j has the maximum x value and thus a state where S holds is reached after $e+n-1$ rounds.

Base case ($e = 0$): Each maximum x value remains the maximum in every round, so the claim is trivially true.

Induction case ($e > 0$): By the induction hypothesis, for all n, each process l at a

distance of at most $e-1$ satisfies $(Sum\, z : z > x.l \wedge (\exists k :: x.k = z) : 1) < N-n$ after $e-1+n$ rounds. (*)

We proceed by induction on n.

Base case ($n = 1$): By (*), there exists a neighbor l of j that satisfies $(Sum\, z : z > x.l \wedge (\exists k :: x.k = z) : 1) < N-1$ after $e-1$ rounds. It follows that j satisfies $(Sum\, z : z > x.j \wedge (\exists k :: x.k = z) : 1) < N-1$ after e rounds.

Induction case ($n > 1$): By (*), there exists a neighbor l of j that satisfies $(Sum\, z : z > x.l \wedge (\exists k :: x.k = z) : 1) < N-n$ after $(e-1)+(n-1)$ rounds. By the nested induction hypothesis, j satisfies $(Sum\, z : z > x.j \wedge (\exists k :: x.k = z) : 1) < N-n-1$ after $e+(n-1)-1$ rounds. It follows that j satisfies $(Sum\, z : z > x.j \wedge (\exists k :: x.k = z) : 1) < N-n$ after $e-1+n$ rounds.

Appendix 2

Lemma 7 *At each state where S holds, for each process j, there is some process k (possibly identical to j) such that $s.j = k$.*

Proof: Process j could either be the root, be a non-root process at an even level, or be a non-root process at an odd level. We consider each case separately:

Case 1 (j is the root process) : Since $even.j$ and $par.j = j$ holds and action 1 of j is disabled, $s.j = fc.j$.

Case 2 (j is a non-root process at even level) : We consider the following two subcases:

- j is not the first child of $par.j$: There exists some process $l (\neq j)$ such that $ns.l = j$ and $even.l$. Since action 5 of l is disabled, $s.j = fc.l$.
- j is the first child of $par.j$: Since $\neg even.(par.j)$ and $fc.(par.j) = j$ hold, and actions 2 and 4 of j are disabled, $(s.j = par.(par.j)) \vee (s.j = ns.(par.j))$.

Case 3 (j is a non-root process at odd level) : We consider the following subcases:

- j is a leaf process : Since $fc.j = j$ by definition, and actions 2 and 4 of j are disabled $(s.j = par.j) \vee (s.j = ns.j)$.
- j is not a leaf process : Consider the last child l of j so that $ns.l = l$ and $par.l = j$. Since action 3 of l is disabled, $s.j = fc.l$. □

Lemma 8 *At each state where S holds, for each up process j, there is some up process k (possibly identical to j) such that $s.k = j$.*

Proof: Again, we consider three cases:

Case 1 (j is the root process) : If j is not a leaf process, then the last child of j, say l, is at an odd level. Since $ns.l = l$ and action 2 of l is disabled, $s.(fc.l) = j$. Else, j is a leaf process and thus $fc.j = j$. Since action 1 of j is disabled, $s.j = j$.

Case 2 (j is at even level) : If j is not a leaf process, then consider last child of j, say l. Since $ns.l = l$ by definition and $\neg even.l$, and action 2 of l is disabled, $s.(fc.l) = j$. Else, j is a leaf process and $fc.j = j$. We consider the following two subcases:

- Process j is not the last child of its parent, i.e. $ns.j \neq j$: Since action 5 of j is disabled, $s.(ns.j) = j$.
- Process j is the last child of its parent, i.e. $ns.j = j$: Since action 3 of j is disabled, $s.(par.j) = j$.

Case 3 (j is at odd level) : If j is not the first child of its parent, then there is a process l such that $ns.l = j$ and $\neg even.l$. Since action 4 of l is disabled, $s.(fc.l) = j$. Else, $fc.(par.j) = j$ and $even.par.j$ holds. We consider the following two subcases:

- $par.j$ is the root : Since action 1 of $par.j$ is disabled, $s.(par.j) = j$.
- $par.j$ is not the root : Then $par.j$ is at even level and $par.(par.j)$ is at an odd level. Since actions 3 and 5 of $par.j$ are disabled, $s.(par.par.j) = j \vee s.(ns.par.j) = j$. □